A Survey of Linear-Programming Guided Branching Parameterized Algorithms for Vertex Cover, with Experimental Results

Tobias Sørensen Urhaug

Master Thesis in Computer Science

May 2015

Department of Informatics

University of Bergen

Norway

Acknowledgements

I would like to thank my supervisor Daniel Lokshtanov for always motivating me and for many great discussions. I always left your office with inspiration and new ideas.

Contents

1 intro

Many interesting problems can be solved by algorithms on graphs. Unfortunately there are many problems which are NP-Hard which means they are practically intractable. Vertex Cover is one such problem. When facing an NP-Hard problem there are different strategies for making it tractable. One such strategy is to confine the possible exponential running time to a function of a parameter alone, so called Fixed-Parameter Tractable(FPT) algorithms.

Vertex Cover was one of the first problems shown to be FPT and has since undergone a lot of research. The most common parameter up to date has been the solution size but recently another above guarantee parameter, the gap between the solution size and an optimal LP solution, has gained more attention. This is both due to the fact that better algorithms for many NP-Hard problems can be achieved through better Vertex Cover above LP algorithms and that an above guarantee parameter can itself yield better FPT algorithms for some instances of Vertex Cover.

In this thesis we will do a survey of three published algorithms for Vertex Cover above LP and present them in a concise way. We will also implement the algorithms to see whether we can gain more insight to them through experiments.

2 Preliminaries

In this chapter we will introduce notation used throughout this thesis and some graph theoretical terms needed to understand the algorithms presented.

2.1 Graphs

A graph is a discrete mathematical structure consisting of points called vertices and lines between points called edges. Graphs are widely used to represent different problems in informatics and also many other fields. The vertices in a graph represents some sort of data and edges shows pairwise relations between different data points in the graph. Edges can also store data relevant to the problem.

An example of how a graph can be used to model a real world problem is how to find the cheapest flight from one airport to another. Here the vertices represent airports and edges between vertices represents that there exist a flight between the two airports. In this problem the edges must also store the price of the given flight and maybe other data like company name, duration etc. We then seek to find the cheapest path in the graph from a given airport to a destination. The graph gives us a good representation of the problem on which we can design algorithms. If price is of less importance

for a customer, but she wants as few transfers as possible or as short duration as possible, we can use the same graph and then tweak the algorithm to find a flight with as few transfers or as short duration as possible.

2.1.1 Formal definition

A graph G = (V,E) is an ordered tuple where V is a set consisting of vertices and E is a set consisting of edges. Each edge is also a tuple, denoted by (u,v), where u and v are vertices in the graph. The vertices u and v are also said to be the end-points of the edge. In some graphs loops connecting a vertex to itself and multiple edges between two vertices are allowed, but for our work we will not need such graphs. An example of a graph needing multiple edges is the cheapest flight problem, as there most likely exists different flights from different companies between airports.

In some problems it is necessary for the edges to have a direction. Graphs which allows directed edges are called directed graphs. An example of a directed graph is a model of the streets and intersections in a city. Every vertex represents an intersection and every edge represents a road between intersections. As many cities have one way streets, these streets must be modeled as a directed edge pointing in the direction in which you are allowed to drive. Formally, in a directed graph each edge is an ordered tuple (u, v), starting in u and pointing towards v. This means that in a directed graph (u, v) is different from (v, u).

We denote the vertices of a graph G by V(G) and the edges by E(G). The size of a graph is measured in both vertices and edges. $|V(G)| = n$ and $|E(G)| = m$ are standard notations also used in this thesis.

2.2 Definitions

The open neighborhood of a node v is the set of vertices adjacent to v. Formally we write N(v) = $\{u \in V(G) : (u, v) \in E(G)\}$ where the degree of v, denoted $deg(v)$, equals the size of its neighborhood. The closed neighborhood N[v] is N(v) $\cup$ {v}. Also sets of vertices can have closed and open neighborhoods. For a subset S $\subseteq$ V(G), N(S) = $\cup_{s \in S} N(s) \setminus S$ is the open neighborhood and N[S] = N(S) $\cup$ S the closed.

We denote the set of edges incident to a vertex v by $\delta(v)$. For a subset S $\in V(G)$, $\delta(S)$ is the set of all edges going out from S. For a directed graph, the set of edges pointing to a vertex v is denoted $\delta^-(v)$ and the set of edges going out from v are $\delta^+(v)$. Analogous, $\delta^-(S)$ and $\delta^+(S)$ are all the edges pointing to and going out from a subset S $\in V(G)$.

The subgraph induced by a vertex set S$\subseteq$ V(G) is the graph restricted to only vertices in S and edges in E(G) where both end-points are in S. We

denote the graph induced by S by G[S]. A subset S which induces a graph without any edges is called an independent set.

A bipartite graph G is a graph where the vertex set V(G) can be partitioned into two partitions, L and R, such that every edge connects a vertex from L to a vertex in R. In other words, both L and R are independent sets.

A matching M in a graph G is a subset $M \subseteq E(G)$ such that no two edges in M share an endpoint. A maximal matching is a matching that can not be augmented by another edge and a maximum matching is a largest obtainable matching in a graph.

A path P in a graph is an ordered set of edges such that every consecutive pair of edges is of the form (u, v), (v, s). An augmenting path relative to a matching M is a path where all the edges alternates between being matched and unmatched. The first and last edges in an augmenting path are unmatched and the first and last vertex in an augmenting path can not be the endpoint of any edge in M. Notice that taking the symmetrical difference of matched and unmatched edges in an augmenting path will augment the matching with one edge. This is why augmenting paths have a crucial role in finding maximum matchings in graphs.

In directed graphs, a subset S $\in V(G)$ is said to be strongly connected if there exists a path in both directions between any pair of vertices in S. A strongly connected component S is said to be tail strongly connected if $\delta^+(S) = \emptyset$.

2.3 P vs NP

In computer science the question whether P equals to NP is a major unsolved problem. The class P is the class of all problems which can be solved in polynomial time while the class NP is the class of all the problems which can be verified in polynomial time.

In NP there is a subset of problems which we call NP-Hard. This is the subset of problems which any other problem in NP can be reduced to in polynomial time. This means that if a polynomial solution is found for one NP-Hard problem, a polynomial solution is found for all.

Even though a huge polynomial like n^{1000} is clearly intractable, these polynomials rarely occur and in practice the divide between P and NP-Hard problems have been a useful measurement for tractability. Complexity theory is a huge research field in Computer Science and this section is just a short introduction to the terminology from it we need for this thesis.

2.4 Vertex Cover

Vertex Cover is a well known NP-Hard problem. In a Vertex Cover instance you are given a graph G and are asked to find a smallest possible subset S of the vertices in G such that all edges in G have at least one end-point in

S. In other words, the vertices in S covers all the edges in the graph. Given a graph G, we denote the size of its minimum obtainable vertex cover by $vc(G)$.

2.5 Parameterized Problems

Some times it makes more sense to formulate NP-Hard problems as parameterized problems. In a parameterized problem you are given both a problem instance and a parameter as input. The parameter is often the size of the desired solution but can also be structural objects like subgraphs of the graph or a matching in the graph. The motivation to add a parameter is that the information it gives can be useful to design smarter and more efficient algorithms. A parameter can also be used to analyze the running time of the algorithm(see section Fixed Parameter Tractable Algorithms below).

Definition 2.1. *A parameterized problem is a language* $L \subseteq \Sigma^* \times \mathbb{N}$ *where* Σ *is a fixed finite alphabet and the second component is called the parameter.*

In the parameterized version of Vertex Cover you are given as input a graph G and an integer k and are asked if there exists a vertex cover of size at most k. Notice that we can still find an optimal solution by iteratively increasing the size of k until we find a solution.

Definition 2.2. *Vertex Cover parameterized by solution size*

Instance: Undirected graph G
Parameter: $k \in \mathbb{N}$
Problem: Does G have a Vertex Cover of size at most k?

Vertex Cover has as mentioned undergone a lot of research and the currently best FPT algorithm for the parameterized version of Vertex Cover runs in $\mathcal{O}(1.2738^k + kn)$[3].

2.6 Fixed-Parameter Tractable Algorithms

When dealing with NP-Hard problems there are different ways of attacking the likely inevitable exponential running time needed to solve the problem. Facing an NP-Hard problem we often still need to find a solution and different approaches can be used. If the optimal solution is not of crucial importance but it is sufficient with something "close" to optimal, we can try to design an approximation algorithm that outputs a solution that is bound to be within an acceptable distance from the optimal solution. This approach trades optimality for efficiency. In other cases we know that we only face certain input instances and this restriction may help us find efficient algorithms for these types of inputs. Kernelization where you cut away easy

parts of the problem before starting the exhaustive search and heuristics are also some approaches tackling NP-Hard problems.

In this thesis we will focus on Fixed Parameter Tractability(FPT) as a way of attacking NP-Hard problems. When designing an FPT algorithm you are given as input a parameterized problem and try to exploit the extra information given by the parameter. It is believed that all NP-Hard problems admits an exponential running time and the goal of an FPT algorithm is to confine this exponential running time to a computational function of the parameter k alone.

Definition 2.3. *A parameterized problem L is said to be Fixed Parameter Tractable if given an instance* (x, k)*, we can in time* $\mathcal{O}(f(k) \cdot n^{\mathcal{O}(1)})$ *decide whether* (x, k) *is in L or not.*

A crucial part in designing FPT algorithms is to find a good parameter. As the running time is decided by a function depending on the parameter alone it makes sense to find parameters that are usually small. A problem can always be attacked from different angles and trying to solve a problem with different parameters in mind gives a deeper understanding of the problem. This can be a motivation by itself for designing FPT algorithms.

FPT algorithms are not necessarily more efficient than exact algorithms. This is because of the unrestricted nature of $f(k)$ which can hide a tower of exponentials like $2^{2^{2^k}}$, a function giving astronomical running times even for small k. The polynomial part of the running time can also be huge. Often research focuses on pushing just one of these factors down, ignoring the other. Some algorithms may be linear in the input size but have a huge f(k) and vice versa.

In this thesis we will compare three different FPT algorithms solving Vertex Cover Above LP to observe how they perform in practice. The first algorithm is the conceptual simplest, running in time $\mathcal{O}(4^\mu m\sqrt{n})$[4]. The second algorithm introduces some more refined reduction rules that improves the exponent in f(μ), giving it a running time of $\mathcal{O}(2.6181^\mu m\sqrt{n})$[8]. The last algorithm focuses on reducing the polynomial factor, giving a running time of $\mathcal{O}(4^\mu(m+n))$[6].

The three algorithms are slight variations of each other. The first giving a simple FPT algorithm while the second and third focuses on improving the exponential part and the polynomial part respectively. It is therefore of great interest to us to see which of these algorithms perform best when implemented. It may be that the theoretical improvements introduced in the second and third algorithms will be lost in practice due to extensive overhead computations during the algorithm. It is also interesting to see which of the exponential and polynomial improvements works better in practice.

3 Kernel based on Linear Programing

In this chapter we will show how to exploit a linear program solution of Vertex Cover to obtain a linear kernel for the problem.

3.1 Kernelization

When looking at parameterized NP-Hard problems we can often in polynomial time find parts of the instance which either can be included in the solution or be ignored and removed. When such parts are found we say that we reduce the given instance to a smaller equivalent one, equivalent meaning that the given instance is a yes-instance if and only if the reduced instance is a yes-instance. This technique allows us to efficiently cut away easy parts of the problem to make the instance is as small as possible before we start the time consuming exponential algorithm.

The way we find such parts are by applying reduction rules. A reduction rule is a mapping that in polynomial time maps a given instance to an equivalent smaller instance. A reduction rule is often dependent on the parameter, but can also be parameter independent. The goal of a reduction rule is to make the instance smaller and it can achieve this by either removing parts of the instance or decrease k, often both.

Definition 3.1. *For a parameterized problem L, a reduction rule is a mapping that in polynomial time maps an instance $(x, k) \in \Sigma^* \times \mathbb{N}$ to an equivalent smaller instance $(x', k') \in \Sigma^* \times \mathbb{N}$. Let $\phi : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ be the mapping, then the following conditions must hold:*

- $(x, k) \in L \iff \phi(x, k) = (x', k') \in L$
- *ϕ can be computed in time polynomial in $|x|$ and k*

We say that a reduction rule is safe or that a reduction rule is sound if it satisfies these conditions.

After finding a set of reduction rules we can combine them to design a kernelization algorithm. A kernelization algorithm takes as input a parameterized instance of a problem and reduces the instance by applying the reduction rules. Sometimes the order in which the reduction rules are applied is important. After all reduction rules are applied exhaustively we may have additional information about the instance that we can exploit algorithmically. For example it can help us prove an upper bound on the size of the reduced instance.

Definition 3.2. *Given an instance $(x, k) \in \Sigma^* \times \mathbb{N}$ and a finite set of reduction rules, a kernelization algorithm for a parameterized problem $L \in \Sigma^* \times \mathbb{N}$ reduces the instance to a problem kernel (x', k') in time polynomial*

in $|x| + k$. *The following conditions must hold for a kernelization algorithm to be sound:*

- $|x'| \leq g(k)$
- $k' \leq h(k)$
- $(x, k) \in L \iff (x', k') \in L$

We call the output of a kernelization algorithm the problem kernel. If $g(k) = k^{\mathcal{O}(1)}$ we say that the problem admits a polynomial kernel. Because we will start the time consuming algorithm on the problem kernel, we try to find as small a kernel as possible. The soundness of the kernel produced by a kernelization algorithm follows directly from the soundness of the reduction rules given.

The classical kernelization example for Vertex Cover is Buss' reduction. This algorithm uses two simple observations about the Vertex Cover problem to remove vertices of high degree and also to remove isolated vertices. A vertex is said to be isolated if it has no edges incident to it. As we are only interested in vertices that can cover edges, the isolated vertices are of no interest and we can remove them from the graph. This proves the soundness of the following reduction rule, which is an example of a parameter independent rule.

Reduction Rule 1. *Remove all isolated vertices in G*

If a vertex v has degree more than k it is easy to verify that v must be in any vertex cover of size at most k. Assume v is not included in the vertex cover, then all its neighbors have to be, in order to cover all edges incident to v. But then we will already have exceeded our budget of k vertices in the vertex cover. This proves the soundness of the following reduction rule, which is an example of a parameter dependent reduction rule:

Reduction Rule 2. *If G has a vertex v of degree more than k, include v in the vertex cover, remove v from G and reduce k by one.*

By applying these two reduction rules we can design a kernelization algorithm for the Vertex Cover problem which gives a quadratic kernel.

Lemma 3.1. *Vertex Cover admits a quadratic kernel where* $|V(G)| \leq k(k+1)$ *and* $|E(G)| \leq k^2$.

Proof. After exhaustively applying reduction rule 1 and 2, we know that the graph has no vertices of degree more than k and no isolated vertices. Assuming there exists a vertex cover of size k, these k vertices can have at most k neighbors each. This means that at most k^2 vertices in G are not

in the vertex cover. This gives a total of $k(k+1)$ vertices in the reduced graph.

All the edges in the reduced graph must also be incident to one of the vertices in the vertex cover. As the maximum degree in the graph is k, the k vertices in the vertex cover can cover at most k^2 edges, which upper bounds the number of edges and proves the quadratic kernel of Vertex Cover.

If the reduction rules have been applied exhaustively and the graph has either more than $k(k+1)$ vertices or more than k^2 edges, we instantly know that there exists no vertex cover of size at most k. □

Lemma 3.1 shows how we can exploit the additional information applying reduction rules gives us to analyze the size of a kernel.

3.2 Integer Linear Programing

In an Integer Linear Programing (ILP) instance we are given a set of variables, a set of linear constraints and a cost function. The goal of an ILP is to find an integer evaluation of the variables such that all constraints are satisfied and the cost function is optimized(maximized or minimized, depending on the problem). Many NP-Hard problems can easily be expressed as integer linear programs. A polynomial time algorithm for ILP would thus imply that P = NP, which is highly unlikely. We therefore have little hope to find efficient algorithms for ILP.

In the ILP formulation for Vertex Cover we create a variable x_v for every vertex v ∈ V(G). If a variable is set to one, it means that the corresponding vertex is in the vertex cover. As this is a vertex cover instance we need to make sure that every edge is covered by at least one vertex. We do this by creating m constraints, one for each edge, s.t. if (u,v) is an edge in the graph then the sum of x_u and x_v must be greater or equal to one. We measure the cost of an ILP solution $\mathbf{x}$ by the sum of all its variables. We denote an ILP solution by ILPVC(G).

Definition 3.3. *Integer Linear Program for Vertex Cover*

$$
\begin{aligned}
&\textit{minimize:} && \sum_{v \in V(G)} x_v && \\
&\textit{subject to:} && x_u + x_v \geq 1 && \forall (u,v) \in E(G) \\
& && 0 \leq x_v \leq 1 && \forall v \in V(G) \\
& && x_v \in \mathbb{Z} && \forall v \in V(G)
\end{aligned}
$$

In a Linear Programming relaxation of Vertex Cover, denoted LPVC(G), the integrality constraint from ILPVC(G) is relaxed so that all variables can

take any real numbered value. Each edge constraint in a LP for Vertex Cover still has to be completely covered, but instead of being covered by one or both end-points taking the value one, an edge can be fractionally covered from both its end-points. For example given an edge (u, v) this edge can be covered by giving both u and v the value one half. This means we take "half of u" and "half of v" in the vertex cover, thus a solution to LPVC(G) will not necessarily give us an optimal vertex cover solution. Nonetheless we gain a lot of information from a LPVC(G) that we can exploit when finding an optimal vertex cover. The real strength of Linear Programs is that they can be solved efficiently in polynomial time. The formal definition for LPVC(G) is:

Definition 3.4. *Linear Program for Vertex Cover*

$$\begin{aligned} &\textit{minimize:} \sum_{v \in V(G)} x_v \\ &\textit{subject to:} x_u + x_v \geq 1 && \forall (u, v) \in E(G) \\ &0 \leq x_v \leq 1 && \forall v \in V(G) \\ &x_v \in \mathbb{R} && \forall v \in V(G) \end{aligned}$$

The solution to a LPVC(G) will be used to design reduction rules in all three algorithms presented in this thesis. How quickly we can compute a LPVC(G) solution in a branching step thus greatly affects the polynomial part of the running time in all three algorithms. It is therefore important to thoroughly understand how solutions to LPVC(G) are computed, especially in order to try to improve the polynomial part of the running time. Lets first fix some notations and some results.

A feasible solution $\mathbf{x} = (x_v)_{v \in V(G)}$ is a vector in $\mathbb{R}^{|V(G)|}$ such that all the constraints of LPVC(G) are satisfied. The cost, often also referred to as the weight of the solution, is denoted by $\mathbf{w(x)} = \sum_{v \in V(G)} x_v$. The value of an optimal solution to LPVC(G) will be denoted by $vc^*(G)$.

Clearly, every feasible solution to an ILPVC(G) is also a feasible solution to the corresponding LPVC(G) relaxation. The optimal weight of an ILPVC(G) instance is therefore an upper bound to the optimal weight of the corresponding LPVC(G) relaxation. As the cost of an optimal solution to ILPVC(G) equals the size of a minimal vertex cover in G, we get the following lemma:

Lemma 3.2. $vc^*(G) \leq vc(G)$

The value $vc^*(G)$ thus gives a lower bound to the size of a vertex cover in G. We will use this later to establish a new parameter on which we can analyze the running time of our algorithms.

Lemma 3.3. *Given an Vertex Cover instance (G,k), if $k < vc^*(G)$, then (G,k) has no vertex cover of size at most k.*

Proof. We know instantly from lemma 3.2 that $k < vc^*(G) \leq vc(G)$, hence the instance is a no instance. □

We can also make some more simple but useful observations about solutions to LPVC(G).

Lemma 3.4. *For a graph G, the vector $\boldsymbol{x}$ where all variables takes value $\frac{1}{2}$ is a feasible solution to LPVC(G).*

Proof. Is it easy to see that $x_u + x_v = \frac{1}{2} + \frac{1}{2} \geq 1$ for every edge (u,v) $\in E(G)$. □

We refer to solutions where all variables take value one half as the all-half solution. We can use the previous lemma to upper bound the value of any optimal solution to LPVC(G)

Corollary 3.5. *For any given graph G, the value $vc^*(G)$ is at most half the number of vertices in the graph.*

$$vc^*(G) \leq \frac{|V(G)|}{2}$$

3.3 Linear Kernel for Vertex Cover

In section 3.1 we showed how to create a quadratic kernel by applying some simple reduction rules that removed isolated vertices and vertices of high degree in the graph. In this section we add a new reduction rule that is based on an optimal LPVC(G) solution. This added rule will improve the kernel size so it becomes linear.

Given an optimal solution $\mathbf{x} = (x_v)_{v \in V(G)}$ of LPVC(G), we can partition the vertices in the graph into three partitions based on the value of their corresponding variable. The intuition behind these partitions is that vertices whose variable have higher fractional value are of bigger importance than the vertices whose variable admits a low value. The partitions are as follows:

Definition 3.5. *Given a graph G and an optimal LPVC(G) solution $\boldsymbol{x}$, we define the following three partitions:*

- $V_0^x = \{v \in V(G) : x_v < \frac{1}{2}\}$
- $V_{\frac{1}{2}}^x = \{v \in V(G) : x_v = \frac{1}{2}\}$
- $V_1^x = \{v \in V(G) : x_v > \frac{1}{2}\}$

From the properties of a feasible solution to LPVC(G) and definition 3.5, we can state some useful facts about V_0 and V_1. In all the coming lemmas we assume $\mathbf{x}$ to be an optimal LPVC(G) solution from which the sets $V_0, V_{\frac{1}{2}}$ and V_1 have been derived.

Lemma 3.6. *V_0 is an independent set.*

Proof. Assume there exist an edge between two vertices u and v in V_0. Then $x_v + x_u < \frac{1}{2} + \frac{1}{2} = 1$, contradicting $\mathbf{x}$ as a feasible solution to LPVC(G). □

Lemma 3.7. *The neighborhood of V_0 equals V_1*

Proof. We show two inclusions to prove the lemma:

- $N(V_0) \subset V_1$
  Assume there exist an edge from a vertex u in V_0 and a vertex v in $V_{\frac{1}{2}}$. Then $x_u + x_v < \frac{1}{2} + \frac{1}{2} = 1$, contradicting $\mathbf{x}$ as a feasible solution to LPVC(G). This proves that all edges out from V_0 goes to vertices in V_1.

- $V_1 \subset N(V_0)$
  Assume there exist a vertex v in V_1 which is not in the neighborhood of V_0. Then all the neighbors of v has a variable with value at least one half. Reducing the value of the variable of v to one half would then still yield a feasible solution to LPVC(G), which is clearly better than $\mathbf{x}$, This contradicts the optimality of $\mathbf{x}$, hence all vertices in V_1 must be the neighbor of a vertex in V_0

□

Lemma 3.8. *If $V_1 < V_0$, there exist a matching M between V_0 and V_1 that saturates V_1.*

Proof. Assume no such matching exist. Then Hall's Theorem [5] guarantees that there exist a subset S of V_1, such that $N(S) < S$. Let ϵ be the smallest distance from a variable in N[S] to one half:

$$\epsilon = min\{\left|x_v - \tfrac{1}{2}\right| : v \in N[S]\}$$

We can now create a new feasible solution $\mathbf{y}$ to LPVC(G):

$$y_v = \begin{cases} x_v - \epsilon & v \in S \\ x_v + \epsilon & v \in N(S) \\ x_v & otherwise \end{cases}$$

As $N(S) < S$, this will be a feasible solution with $\mathbf{w(y)} < \mathbf{w(x)}$, contradicting the optimality of $\mathbf{x}$. □

The classic Nemhauser-Trotter Theorem shows how these partitions can be used to reduce an Vertex Cover instance.

Theorem 3.9. *(Nemhauser-Trotter's Theorem) [9]*
Given a graph G and an optimal LPVC(G) solution $\boldsymbol{x}$ *partitioning G into* V_0^x, $V_{\frac{1}{2}}^x$ *and* V_1^x, *there exists a minimum vertex cover S such that S contains all of* V_1^x *and none of* V_0^x

$$V_1^x \subseteq S \subseteq V_{\frac{1}{2}}^x \cup V_1^x$$

Proof. Let S^* be an optimal vertex cover for G. We show that we can construct an optimal vertex cover S from S^* by removing all vertices in V_0^x from S^* and adding all vertices in V_1^x to S^*:

$$S = (S^* \setminus V_0^x) \cup V_1^x$$

Given the way the partitions are created, every vertex in V_0^x must have an edge to a vertex in V_1^x. From this it is easy to see that S is a valid vertex cover of G. We now have to prove that S is also optimal. To get to a contradiction, assume the contrary, that $|S| > |S^*|$. We already know the size of S:

$$|S| = |S^*| - |V_0^x \cap S^*| + |V_1^x \setminus S^*| \tag{1}$$

Given that our assumption holds, this infers that:

$$|V_0^x \cap S^*| < |V_1^x \setminus S^*| \tag{2}$$

Let us now define an epsilon, which intuitively measures the smallest distance from the value of a variable in $V_0^x \cup V_1^x$ to one half on the real number line.

$$\epsilon = min\{\left|x_v - \tfrac{1}{2}\right| : v \in V_0^x \cup V_1^x\}$$

With this epsilon we can now define a new vector $\mathbf{y} = (y_v)_{v \in V(G)}$ where:

$$y_v = \begin{cases} x_v - \epsilon & v \in V_1^x \setminus S^* \\ x_v + \epsilon & v \in V_0^x \cap S^* \\ x_v & otherwise \end{cases}$$

In our chase for a contradiction we need to show that $\mathbf{y}$ is a feasible solution to LPVC(G) of weight less than $\mathbf{x}$. From the construction of $\mathbf{y}$ and from (2), it is easy to see that $\mathbf{w}(\mathrm{y}) < \mathbf{w}(\mathrm{x})$.

Observe that all variables in $\mathbf{y}$ will take legal values and the partitions with respect to $\mathbf{y}$ will be equal to the partitions with respect to $\mathbf{x}$, due to the construction of ϵ and $\mathbf{y}$.

$$0 \leq (y_v)_{v \in V(G)} \leq 1.$$

For an arbitrary edge (u,v) in E(G), we need to prove that $y_u + y_v \geq 1$. If none of u or v are in $V_1^x \setminus S^*$, then $y_u \geq x_u$ and $y_v \geq x_v$ which implies $y_u + y_v \geq x_u + x_v \geq 1$. We need to consider the case where exactly one of u or v is in $V_1^x \setminus S^*$. By symmetry we can assume u to be this vertex. Then v must be in S^* in order to cover the edge (u,v). This can happen in two ways:

- $v \in V_0^x \cap S^*$
  $\Rightarrow$
  $y_u + y_v = x_u - \epsilon + x_v + \epsilon = x_u + x_v \geq 1$
- $v \in (V_{\frac{1}{2}}^x \cup V_1^x) \cap S^*$
  Notice that $y_u = x_u - \epsilon \geq \frac{1}{2}$ and that $x_v \geq \frac{1}{2}$
  $\Rightarrow$
  $y_u + y_v \geq x_u - \epsilon + x_v \geq \frac{1}{2} + \frac{1}{2} = 1$

We have shown that given the assumption, we can construct a new solution $\mathbf{y}$ to LPVC(G) which is better than $\mathbf{x}$. This is a contradiction which proves that $|S| \leq |S^*|$ which again proves the theorem. □

From Theorem 3.9 and lemmas 3.6 and 3.7 we can now prove that there always exists an optimal solution to LPVC(G) where all the variables in the solution admits values from the set $\{0, \frac{1}{2}, 1\}$. We will refer to such solutions as half-integral optimal solutions to LPVC(G):

Lemma 3.10. *For any optimal solution $\boldsymbol{x}$ to LPVC(G), there exists an optimal half-integral solution $\boldsymbol{x}$'.*

Proof. Let $\mathbf{x}$ be an optimal solution to LPVC(G) and let $\mathbf{x}$' be a vector in which all variables in $\mathbf{x}$ have been rounded to its nearest half-integral value. The weight of $\mathbf{x}$' will then be:

$$\begin{aligned}
\sum_{v \in V(G)} x'_v &= \sum_{v \in V_0 \cup V_1} x'_v + \sum_{v \in V_{\frac{1}{2}}} x'_v \\
&= \sum_{v \in V_0 \cup V_1} x'_v + \sum_{v \in V_{\frac{1}{2}}} x_v \\
&= |V_1| + \sum_{v \in V_{\frac{1}{2}}} x_v \\
&\leq \sum_{(u,v) \in M} (x_u + x_v) + \sum_{v \in V_{\frac{1}{2}}} x_v \\
&\leq \sum_{v \in V_0 \cup V_1} x_v + \sum_{v \in V_{\frac{1}{2}}} x_v = \sum_{v \in V(G)} x_v
\end{aligned} \tag{3}$$

where the first inequality is a result of lemma 3.8 which says there is a matching between V_0 and V_1 saturating V_1.

□

From this point on, whenever we speak about an optimal solution to LPVC(G) we will refer to an optimal half-integral solution where all the variables admits values from the set $\{0, \frac{1}{2}, 1\}$.

A vector in which all variables admits the same value i is denoted $\mathbf{x} \equiv i$, where $i \in \{0, \frac{1}{2}, 1\}$. Of special importance to us is the solution in which all variables takes the value one half. This will be referred to as the all-half solution.

3.3.1 Reduction

Theorem 3.9(Nemhauser-Trotter) gives us the following reduction rule:

Reduction Rule 3. *For a parameterized Vertex Cover instance (G,k) and an optimal solution $\boldsymbol{x}$ to LPVC(G), define V_0^x, $V_{\frac{1}{2}}^x$ and V_1^x as in definition 3.5.*

If $\boldsymbol{w(x)} > k$, lemma 3.3 proves that the given instance is a no-instance. Otherwise, Theorem 3.9(Nemhauser-Trotter) proves that there exists a vertex cover of minimum size that includes all of V_1^x and none of V_0^x. The reduction rule therefore includes V_1^x in the vertex cover and deletes $V_0^x \cup V_1^x$ from the graph. The reduced instance $G' = G[V_{\frac{1}{2}}^x]$ is then a smaller equivalent instance with $k' = k - |V_1^x|$.

The following lemma proves the soundness of reduction rule 3:

Lemma 3.11. *Given an instance (G, k) and an optimal solution $\boldsymbol{x}$ to LPVC(G). Let $G' = G[V_{\frac{1}{2}}^x]$ and $k' = k - |V_1^x|$ be the kernel produced by reduction rule 3. Then (G, k) is a yes-instance if and only if (G', k') is.*

Proof. If (G, k) is a yes instance, let S be a minimum vertex cover of G such that $V_1^x \subseteq S \subseteq V_{\frac{1}{2}}^x \cup V_1^x$. Theorem 3.9(Nemhauser-Trotters Theorem), proves that there exists such a minimum vertex cover. Any vertex cover restricted to a subgraph will always be a vertex cover of this subgraph. This means $S \cap V_{\frac{1}{2}}^x$ will be a vertex cover of G' of size at most $k - |V_1^x|$, proving that if (G, k) is a yes-instance then (G', k') also is.

In the other direction, let S' be a vertex cover of $G[V_{\frac{1}{2}}^x]$ of size k'. As all vertices in V_0^x can only be adjacent to vertices in V_1^x, $S' \cup V_1^x$ will be a vertex cover of G. $|S' \cup V_1^x| = k' + |V_1^x| \leq k - |V_1^x| + |V_1^x| = k$. □

The soundness of reduction rule 3 thus follows from the correctness of the Nemhauser-Trotter theorem. After reduction rule 3 has been applied we are left with a kernel consisting of the graph induced by $V_{\frac{1}{2}}$. When removing the

vertices in $V_0 \cup V_1$ from G, we can also remove the corresponding variables from the optimal LPVC(G) solution **x**. We will now show that not only is this restriction of the LPVC(G) solution a feasible solution to LPVC($G[V_{\frac{1}{2}}]$), it is also optimal.

Lemma 3.12. *Given an instance (G,k) and an optimal solution* $\boldsymbol{x}$ *to LPVC(G). Let* (G', k') *be the new instance after applying reduction rule 3 and let* $\boldsymbol{x}$**'** *be the restriction of* $\boldsymbol{x}$ *after removing all variables corresponding to vertices that have been removed. Then* $\boldsymbol{x}$**'** *will be an optimal solution to LPVC(G').*

Proof. Notice that all edges connecting the vertices in G' and $V_0 \cup V_1$ are covered when the variables corresponding to vertices in V_1 are set to one. Hence, all feasible solutions **y** to LPVC(G') with the added variables corresponding to vertices in $V_0 \cup V_1$ set to zero and one respectively will be a feasible solution to LPVC(G).

Now consider the case where there exists an optimal solution **y** to LPVC(G') which is better than the restriction **x'** found after applying the reduction rule. Then we could get a new feasible solution **x"** to LPVC(G) by adding the variables corresponding to vertices in $V_0 \cup V_1$ set to zero and one respectively, to **y**. This solution will clearly be a solution to LPVC(G) of weight less than $\mathbf{w(x)}$, contradicting the optimality of **x**. Thus proving the optimality of **x'**. □

We now have the results to prove the new improved linear kernel to Vertex Cover:

Theorem 3.13. *Vertex Cover admits a kernel of size at most 2k*

Proof. Given an instance (G,k), let (G', k') be the reduced instance after applying reduction rule 3 according to an optimal LPVC(G) solution **x**. Lemma 3.12 now guarantees that the restriction **x'** of **x** to G' will be an optimal solution where all variables take value one half. This gives us:

$$\left|G[V_{\frac{1}{2}}]\right| = 2\sum_{v \in V_{\frac{1}{2}}} x'_v \leq 2\sum_{v \in V(G)} x_v \leq 2k$$

□

3.4 Efficient Algorithm for LPVC(G)

When computing the kernel in theorem 3.13 we need an efficient way of computing the linear program values of an optimal LPVC(G) solution. We do this be reducing LPVC(G) to finding a minimum vertex cover in a bipartite graph H constructed from G. We construct the vertices in H by making two partitions, L_H and R_H, both being copies of the vertex set V(G).

Definition 3.6. *A bipartite graph* H_G *with partitions* L_H *and* R_H *is constructed from a graph G as follows:*

1. *For every vertex $v \in V(G)$, create two copies v_l and v_r and place them in L_H and R_H in $V(H_G)$ respectively*

2. *For every edge $(u, v) \in E(G)$, create (u_l, v_r) and (u_r, v_l) in $E(H_G)$*

Whenever it is clear from the context which graph H is constructed from we leave the subscript G out and only write H.

3.4.1 Hopcroft-Karp

To find a minimum vertex cover in H we first use Hopcroft-Karp's [7] algorithm to find a maximum matching in H. Like many other algorithms finding maximum matchings, the Hopcroft-Karp algorithm is based on finding augmenting paths. An augmenting path relative to a matching M is a path in which the edges alternates between beeing in the matching M and not. In an augmenting path P the first and last vertices must be free, where a vertex v is said to be free relative to a matching M if none of the edges in M are adjacent to v. That the first and last vertices are free means that the first and last edges in P can not be in M. When having an augmenting path P we can remove all matched edges in P from M and add all unmatched edges in P to M, in order to create a new matching M' with cardinality $|M| + 1$. This is why augmenting paths are used to find maximum matching. When the edges of a path alternates between being and not being in M, but is not an augmenting path, we call it an alternating path.

To formalize the importance of augmenting paths we state the well known fact that a matching M is maximum in a graph G if and only if there is no augmenting path in G.

Lemma 3.14. *(Berge's lemma) [2]*
A matching M is maximum in G $\iff$ there exists no augmenting path in G

Instead of just finding one augmenting path in each iteration of the algorithm, Hopcroft-Karp finds a set of disjoint shortest augmenting paths. Then, instead of augmenting and increasing the size of the matching with just one per iteration, it augments a set of augmenting paths, reducing the number of iterations needed for the algorithm to terminate from $\mathcal{O}(n)$ to $\mathcal{O}(\sqrt{n})$.

In order to find a set of disjoint shortest augmenting paths the algorithm partitions the graph into layers P. The free vertices in L_H form the first layer in P and the starting point for the partitioning. From this layer we perform a Breadth-First Search(BFS) that is guided so it only uses unmatched edges when going from L_H to R_H and only matched edges when going in the other direction. When the BFS encounters a free vertex in R_H, a shortest augmenting path has been found. The BFS finishes finding all vertices reachable in this layer before it terminates.

The algorithm then proceeds to perform a Depth-First Search(DFS) from the free vertices in the last layer in P. The DFS uses the layers in P as guidance, meaning it only traverses edges from a layer to the layer above. When a free vertex in L_H is reached, this will be an augmenting path. The DFS proceeds until it finds a maximal set of disjoint augmenting paths. These paths are then augmented and the algorithm will start again until no more augmenting paths can be found.

It can be proven that the algorithm only needs $\mathcal{O}(\sqrt{n})$ iterations before it terminates. Each iteration, consisting of a BFS and a DFS, can be done in $\mathcal{O}(m+n)$ time, which gives the total running time $\mathcal{O}(m\sqrt{n})$. For the proof of correctness and the running time analysis of Hopcroft-Karp the reader is referred to [7].

3.4.2 Finding minumum vertex cover from maximum matchings in bipartite graphs

We now want to find a minimum vertex cover in H from a maximum matching M in H. Unless M is a perfect matching, where a perfect matching is a matching where all vertices in one of the partitions of a bipartite graph are endpoints of an edge in M, we have free vertices in both L_H and R_H. Let Z be all the vertices reachable in H by an alternating path from the free vertices in L_H. The following lemma shows how to construct a minimum vertex cover from M and Z:

Lemma 3.15. *Let M be a maximum matching in H and let Z be all the vertices reachable by an alternating path in H from the free vertices in L_H. Then*

$$S = (R_H \cap Z) \cup (L_H \setminus (L_H \cap Z))$$

will be a minimum vertex cover for H.

Proof. Notice that there can not exist any edges between $L_H \cap Z$ and $R_H \setminus Z$ as this would imply there exist an augmenting path in H, contradicting the optimality of M. Thus S have to be a valid vertex cover for H.

The size of S has to be the same as the size of M, proving that S is an optimal vertex cover. □

From a minimum vertex cover S of H we can find an optimal solution to LPVC(G):

Lemma 3.16. *Given a graph G and a minimum vertex cover S for H_G, then the vector $\boldsymbol{x}$:*

$$x_v = \begin{cases} 0 & \textit{if none of } v_1 \textit{ or } v_2 \textit{ are in } S \\ \frac{1}{2} & \textit{when exactly one of } v_1 \textit{ or } v_2 \textit{ is in } S \\ 1 & \textit{if both } v_1 \textit{ and } v_2 \textit{ are in } S \end{cases}$$

will be an optimal LPVC(G) solution.

Proof. To show that $\mathbf{x}$ is a feasible solution we need to show that the constraints of LPVC(G) are satisfied. Hence for every edge $(u, v) \in E(G)$ we need to show that $x_u + x_v \geq 1$. As S is a vertex cover of H, we know that for every edge (u, v) in E(G), at least two of the vertices $\{u_1, u_2.v_1, v_2\}$ must be in S. Then either both x_u and x_v will take the value one half or at least one of them will take the value one. This clearly satisfies the constraint $x_u + x_v \geq 1$.

In order to prove the optimality of $\mathbf{x}$, assume $\mathbf{y}$ is an optimal solution to LPVC(G) and show that the weight of $\mathbf{x}$ can be at most the weight of $\mathbf{y}$. Give each vertex v_i $i \in \{1, 2\}$ in H the weight y_v. Then these weights will form a fractional vertex cover of H and we have:

$$\textstyle\sum_{v \in V(G)} y_v = \frac{1}{2} \sum_{v \in V(G)} (w(v_1) + w(v_2))$$

We know that the size of a vertex cover is at least as big as the size of a fractional vertex cover. We also know that the weight of $\mathbf{x}$ equals half the size of S, as each vertex in S will contribute one half to a variable in LPVC(G). This gives us:

$$\begin{aligned} \textstyle\sum_{v \in V(G)} y_v = \frac{1}{2} \sum_{v \in V(G)} (w(v_1) + w(v_2)) = \frac{1}{2} \sum_{v \in V(H)} w(v) \\ \geq \frac{|S|}{2} = \textstyle\sum_{v \in V(G)} x_v \end{aligned}$$

which proves the optimality of $\mathbf{x}$.

□

3.4.3 Algorithm

We now have the results to present the algorithm for finding optimal solutions to LPVC(G). First construct the bipartite graph H from G, then, using Hopcroft-Karp, find a maximum matching M in H. From M and lemma 3.15 we can find a minimum vertex cover S in H. From S and lemma 3.16 we can compute an optimal solution $\mathbf{x}$ of LPVC(G). As Hopcroft-Karp is the time dominating part of the algorithm, we get a $\mathcal{O}(m\sqrt{n})$ running time:

Theorem 3.17. *Given a graph G, we can compute an optimal solution of the linear program LPVC(G) in $\mathcal{O}(m\sqrt{n})$time.*

Theorem 3.17 gives rise to the following corollary:

Corollary 3.18. *We can compute a 2k kernel for Vertex Cover of Theorem 3.13 in $\mathcal{O}(m\sqrt{n})$ time.*

4 Vertex Cover Above LP

In this chapter we will introduce the main problem that we focus on in this thesis, Vertex Cover Above LP.

4.1 Above Guarantee Parameterization

Some parameterized problems have a natural lower bound to their solution size. If this lower bound is relatively big, then using the solution size as parameter might not be the most appropriate. The idea behind FPT algorithms is to make NP-Hard problems tractable by confining the exponential running time to a function of a parameter k alone. The running time of an FPT algorithm can be expressed as $f(k) \cdot n^c$, where $f(k)$ is a (possibly) exponential function depending on k alone. In practice FPT algorithms will only be tractable for small k. It will therefore make sense to search for a different parameter in situations where there is a relatively big lower bound to the solution size.

One approach is to use the excess above the lower bound as a parameter. We still ask if the problem has a solution of size k, but we analyze the algorithm with the parameter $\mu = k - l$, where l is the size of the lower bound. By using this parameter we obtain a running time of $\mathcal{O}(f(k-l) \cdot n^{\mathcal{O}(1)})$. This parameterization, called Above Guarantee Parameterization, makes sense as the parameter will be much smaller than k. Depending on the exponent in the running time and how efficiently the lower bound can be computed, this can give more efficient algorithms.

Vertex Cover is a problem which has a natural lower bound in the size of the maximum matching in the given graph. A maximum matching M in a graph is a collection of edges where no two edges in M share an end-point. In order to cover all these edges, the vertex cover has to include at least one end-point from every edge. The size of M therefore gives a lower bound to the size of a vertex cover in a graph. This parameter gives rise to an important problem called Vertex Cover above Matching.

Definition 4.1.
Vertex Cover Above Matching

Input: Undirected graph G, integer k
Parameter: $\mu(G, k) = k - |M|$
Problem: Does G have a vertex cover of size at most k?

This is an important problem because many interesting problems like ALMOST-2-SAT and ODD CYCLE TRAVERSAL can be reduced to it. Improved algorithms for Vertex Cover above Matching can therefore also give better algorithms for these problems. The interested reader is referred to [8] for a more detailed introduction to the problem and its reduction to other problems.

4.2 Vertex Cover Above LP

As already mentioned, Vertex Cover has a natural lower bound in the size of the maximum matching of a graph. In section 3.2 we also got introduced to another lower bound for Vertex Cover, the optimal linear program value $vc^*(G)$. With this lower bound we can introduce a new problem Vertex Cover above LP, which is the problem this thesis will focus on:

Definition 4.2.
Vertex Cover Above LP

Input: Undirected graph G, integer k
Parameter: $\mu(G, k) = k - vc^*(G)$
Problem: Does G have a vertex cover of size at most k?

It is easy to see that $vc^*(G) \geq |M|$, where M is a maximum matching in G, as every edge in M has to be fractionally covered by the variables in any solution to LPVC(G). The parameter $k - vc^*(G)$ in Vertex Cover above LP will therefore always be at most as big as the parameter in Vertex Cover above Matching:

$$k - vc^*(G) \leq k - |M|$$

Due to this, every parameterized algorithm for Vertex Cover above LP will also be a parameterized algorithm for Vertex Cover above Matching. Improved algorithms for Vertex Cover above LP will therefore give improved algorithms to all the problems which currently best algorithms are reductions to Vertex Cover above Matching.

5 The Surplus of a vertex set

Up until this point we have viewed the sets V_0 and V_1 found by reduction rule 3 from a linear programing point of view. They have been the vertices in which the corresponding variables admit integer values in an optimal LPVC(G) solution. It is also possible to look at these sets from a graph structural perspective which can help further understanding the reduction rules and algorithms presented.

Reduction rule 3 guarantees that it is safe to remove the vertices in V_0 and V_1 from the graph when all the vertices in V_1 are included in the vertex cover. That it is safe to remove all the vertices in V_0 without including any of them in the vertex cover means that there can not be any edges between any vertices in V_0, i.e. V_0 is an independent set.

Proposition 5.1. *The vertices in* V_0 *found by reduction rule 3 form an independent set.*

Further, since it is safe to only include vertices in V_1 into the vertex cover in order to also remove V_0 from the graph, we know that all outgoing edges from V_0 are covered by vertices in V_1. This means that the neighborhood of V_0 has to be included in V_1.

Proposition 5.2. *Let V_0 and V_1 be the sets found by reduction rule 3. Then $N(V_0) \subseteq V_1$*

Hence we see that what reduction rule 3 is really searching for are independent sets where it is guaranteed that including the neighborhood of the independent set into the vertex cover will give a solution at least as good as any optimal solution. To further formalize this idea we introduce the notion of surplus for independent sets. The surplus of an independent set is defined to be the difference between the size of its neighborhood and itself.

Definition 5.1. *The surplus of an independent set Z, denoted by **surplus**(Z), is $|N(Z)| - |Z|$*

To further expand on this idea, the surplus of a graph is defined to be the minimum surplus of any independent set in the graph. This is an important concept which we will use later when analyzing the algorithms.

Definition 5.2. *Let $\mathcal{A}$ be the set of all independent sets in a graph G. The surplus of the graph, denoted **surplus**(G), is defined to be the minimum surplus over all sets in $\mathcal{A}$*

$$\textbf{\textit{surplus}}(G) = \min_{X \in \mathcal{A}} \textbf{\textit{surplus}}(X)$$

Translated to the notion of surplus, reduction rule 3 finds independent sets with negative surplus and removes them from the graph. Intuitively, as independent sets with negative surplus has a neighborhood of lesser cardinality that covers all edges between the independent set and the neighborhood, it is tempting to generalize reduction rule 3 so that all neighborhoods of independent sets with negative surplus can safely be included in any optimal vertex cover. However, care must be taken as there exist examples of independent sets with negative surplus where the optimal solution does not include all of its neighborhood.

The independent sets V_0 found by reduction rule 3 have an extra property which makes it safe to include $N(V_0) = V_1$ in the vertex cover and remove $N[V_0]$ from the graph. They are inclusion-minimal sets Z with negative surplus, meaning that for any subset Z' of Z the surplus of Z' is greater or equal to the surplus of Z:

$$\textbf{surplus}(Z') \geq \textbf{surplus}(\mathrm{Z}), \text{ for all } Z' \subseteq Z$$

This means that reduction rule 3 finds inclusion-minimal independent sets of negative surplus and removes them from the graph. Thus after reduction rule 3 has been applied, $\mathbf{x} \equiv \frac{1}{2}$ is an optimal solution to LPVC(G) and the graph G has a surplus of at least zero:

Lemma 5.1. *Let* $\boldsymbol{x} \equiv \frac{1}{2}$ *be an optimal solution to LPVC(G), then*

$$\boldsymbol{surplus}(G) \geq 0$$

Proof. Assume G has an independent set Z with negative surplus. Let x' be a new vector with variables set to the following values:

$$x'_v = \begin{cases} x_v & v \in V(G) \setminus N[Z] \\ 0 & v \in Z \\ 1 & v \in N(Z) \end{cases}$$

clearly x' will be a feasible solution to LPVC(G) with weight

$$\mathbf{w}(x') = \mathbf{w}(\mathrm{x}) + \tfrac{1}{2}\ (|N(Z)| - |Z|)$$

as the surplus of Z is negative, $(|N(Z)| - |Z|)$ must be negative. Thus

$$\mathbf{w}(x') < \mathbf{w}(\mathrm{x})$$

which contradicts the optimality of x. Hence the surplus of G is greater or equal to zero. □

The following two lemmas give rise to an even stronger and more general result:

Lemma 5.2. *For any independent set Z with surplus i in G, the vector* x^Z *where:*

$$x^Z_v = \begin{cases} 0 & v \in Z \\ 1 & v \in N(Z) \\ \frac{1}{2} & v \in V(G) \setminus N[Z] \end{cases}$$

will be a feasible solution to LPVC(G) with weight $\boldsymbol{w}(x^Z) = \frac{|V(G)|}{2} + \frac{i}{2}$.

Proof. x^Z is clearly a feasible solution to LPVC(G). To prove the weight of x^Z, set all variables to one half, then subtract one half for each vertex in Z as these are now set to zero and add one half for each vertex in N(Z) as these are now set to one. This gives

$$\tfrac{|V(G)|}{2} + \tfrac{1}{2}(|N(Z)| - |Z|) = \tfrac{|V(G)|}{2} + \tfrac{1}{2}(\mathbf{surplus}(\mathrm{Z})) = \tfrac{|V(G)|}{2} + \tfrac{i}{2}$$

which proves there exist a solution $\mathbf{x}$ to LPVC(G) with the desired weight. □

The converse of this lemma is indeed also true:

Lemma 5.3. *For any half-integral solution* $\boldsymbol{x}$ *of LPVC(G),* V_0 *will be an independent set with surplus* $i \leq (\boldsymbol{w(x)} - \frac{|V(G)|}{2}) \cdot 2$

Proof. V_0 is an independent set by lemma 3.6. To prove the surplus of V_0, observe that $N(V_0)$ must be a subset of V_1. The weight of $\mathbf{x}$ is:

$$\mathbf{w(x)} = \frac{|V(G)|}{2} + \frac{1}{2}(|V_1| - |V_0|)$$

As $N(V_0) \subseteq V_1$, $|V_1| - |V_0| \geq \mathbf{surplus}(V_0)$. This gives us:

$$\mathbf{w(x)} = \frac{|V(G)|}{2} + \frac{1}{2}(|V_1| - |V_0|) \geq \frac{|V(G)|}{2} + \frac{1}{2}\ \mathbf{surplus}(V_0).$$

rewriting this we get:

$$\mathbf{surplus}(V_0) \leq 2{\cdot}\mathbf{w(x)} \text{ - } |V(G)|$$

proving the lemma. □

We can now prove the following useful corollary:

Corollary 5.4. $vc^*(G) = \frac{|V(G)|}{2} + \frac{1}{2}(\boldsymbol{surplus}(G))$

Proof. We prove the corollary in two parts:

- $vc^*(G) \leq \frac{|V(G)|}{2} + \frac{1}{2}(\mathbf{surplus}(\mathrm{G}))$
  Let Z be an independent set of minimum surplus in G, then x^Z from lemma 5.2 will be a feasible solution to LPVC(G) with $\mathbf{w}(x^Z) = \frac{|V(G)|}{2} + \frac{1}{2}\mathbf{surplus}(\mathrm{G})$, proving the inequality.

- $vc^*(G) \geq \frac{|V(G)|}{2} + \frac{1}{2}(\mathbf{surplus}(\mathrm{G}))$
  Let $\mathbf{x}$ be an optimal half integral solution to LPVC(G), then lemma 5.3 guarantees that $vc^*(G) = \mathbf{w(x)} \geq \frac{|V(G)|}{2} + \frac{1}{2}\ \mathbf{surplus}(V_0^x) \geq \frac{|V(G)|}{2} + \frac{1}{2}(\mathbf{surplus}(\mathrm{G}))$

□

6 Simple Algorithm for Vertex Cover above LP

We will now describe in detail the first and conceptually simplest of the three algorithms for Vertex Cover Above LP that we have implemented. This algorithm is based on the algorithm given in [4] and will also form the basis for the next two improved algorithms. The algorithm exhaustively searches for a vertex cover of size at most k by first applying a set of reduction rules before branching using a familiar neighborhood branching strategy for vertex cover. Even though the algorithm is quite simple, analyzing the running time using the above guarantee parameter $\mu(G, k)$ is more advanced and it is of great interest to us to understand how this parameter behaves during the algorithm.

In the following sections we will present the algorithm. We first show the reduction rule used, prove its correctness and show how applying it affects the parameter. We then proceed to describe the branching strategy and also

show how it affects the parameter $\mu(G, k)$. We then wrap up to give the complete algorithm and prove its running time.

6.1 Reduction

In the first stage of the algorithm we want to reduce the graph to as small an instance kernel as possible. In section 3.3 we showed how to obtain a linear kernel based on reduction rule 3. Here we will use a stronger version of that rule so that exhaustively applying it will not only give us a 2k kernel, but also guarantee that the all-half vector is the unique optimal LP solution to the kernel. The all-half uniqueness is important for the running time analysis of this algorithm.

Lemma 6.1. *For a graph G with no isolated vertices, an optimal solution to LPVC(G) which is not the all-half solution can be found in $\mathcal{O}(mn^{\frac{3}{2}})$ time. If no such solution can be found the all-half solution is the unique optimal solution to LPVC(G).*

Proof. First compute a solution $\mathbf{x}$ to LPVC(G) using the algorithm from theorem 3.17 in section 3.4. If $\mathbf{x}$ is not the all-half solution simply return it. If $\mathbf{x} \equiv \frac{1}{2}$, proceed as follows:

To find an optimal non all-half solution we try, one by one, to fix a variable to be one. If the optimal LP value does not change after the variable x_v is fixed to be one, this will be a feasible solution with at least one integer value, namely x_v.

For every vertex v ∈ V(G), solve the Linear Program where v is removed from the graph. Let x^v be an optimal solution to LPVC(G-v) with value $vc^*(G-v)$. If $vc^*(G-v) \leq vc^*(G) - 1$, fix x_v to be one and let $x^{v,0}$ be x^v with the added variable x_v. This will be a feasible solution to LPVC(G), as adding a variable with value one surely covers all edges incident to it. Given the assumption it will also be an optimal solution that is not the all-half solution. Return $x^{v,0}$.

If no vertex in G gives us an optimal non all-half solution we need to prove that the all-half solution indeed is the unique optimal solution. That no vertex in G gave an optimal non all-half solution means that:

$$vc^*(G-v) > vc^*(G) - 1 \text{ for all v} \in \text{V(G)}$$

Notice, as there exist an half-integral optimal solution for any optimal solution, see lemma 3.10, this slightly stronger assumption also holds:

$$vc^*(G-v) \geq vc^*(G) - \tfrac{1}{2} \text{ for all v} \in \text{V(G)}$$

To prove the uniqueness of the all-half solution we chase a contradiction. Assume $\mathbf{x}$ is an optimal solution to LPVC(G) that is not the all-half solution. As we already know that the all-half solution is optimal, there must exist

a vertex v ∈ V(G) s.t. $x_v > \frac{1}{2}$. Then x restricted to G-v will be a feasible solution to LPVC(G-v) with value:

$$\mathbf{w(x)} - x_v = vc^*(G) - x_v < vc^*(G) - \tfrac{1}{2}$$

which is a contradiction that proves the uniqueness of the all-half solution. The running time can easily be seen as we in worst case have to invoke the algorithm from Theorem 3.17 n+1 times. □

We can now state our enhanced reduction rule:

Reduction Rule 4. *Invoke the algorithm from lemma 6.1. If an optimal solution* $\boldsymbol{x}$ *is returned and* $\boldsymbol{x}$ *is not the all-half solution, use reduction rule 3 to reduce the instance according to* $\boldsymbol{x}$*. If* $\boldsymbol{x} \equiv \frac{1}{2}$*, we know it is the unique optimal solution and can start to branch on the kernel* $G[V_{\frac{1}{2}}]$*.*

Once Reduction Rule 4 is not longer applicable, we know that we have a problem kernel G' where the all-half vector is the unique optimal solution to LPVC(G'). The uniqueness of the all-half vector as an optimal solution gives us further information about the surplus of the graph. The following lemma builds on 5.1 and strengthens it according to the knowledge that $\mathbf{x} \equiv \frac{1}{2}$ is the unique optimal solution to LPVC(G).

Lemma 6.2. *Given a graph G where* $\boldsymbol{x} \equiv \frac{1}{2}$ *is the unique optimal solution, then the surplus of G is at least one:*

$$\boldsymbol{surplus}(G') \geq 1$$

Proof. Assume there exists an independent set Z with $\mathbf{surplus}(Z) = 0$. Let x' again be a new vector with variables set to the following values:

$$x'_v = \begin{cases} x_v & v \in V(G) \setminus N[Z] \\ 0 & v \in Z \\ 1 & v \in N(Z) \end{cases}$$

clearly x' will be a feasible solution to LPVC(G). Lemma 5.2 guarantees that:

$$\mathbf{w}(x') = \tfrac{|V(G)|}{2} + \tfrac{1}{2}\mathbf{surplus}(Z) = \tfrac{|V(G)|}{2}$$

which contradicts the unique optimality of $\mathbf{x}$. Hence G must have a surplus of at least one. □

The algorithm will be analyzed using the measure $\mu(G, k)$ and it is therefore of importance to understand how the reduction rule influences this measure.

Lemma 6.3. *Let an instance (G', k') be created by applying Reduction Rule 3 to an optimal LPVC(G) solution $\boldsymbol{x}$. Then $\mu(G', k') = \mu(G, k)$. In other words, the value of the new parameter will equal the value of the old.*

Proof. In order to prove the statement it is sufficient to prove the following two equalities:

$$vc(G) = vc(G') + |V_1^x|$$
$$vc^*(G) = vc^*(G') + |V_1^x|$$

as they are implying:

$$vc^*(G) - vc^*(G') = vc(G) - vc(G') = |V_1^x| = k - k'$$
$$k - vc^*(G) = k' - vc^*(G')$$
$$\mu(G, k) = \mu(G', k')$$

- $vc(G) = vc(G') + |V_1^x|$
  - $vc(G) \leq vc(G') + |V_1^x|$
    For every vertex cover S of G', adding V_1 to S will give a feasible vertex cover to G as all vertices in V_0 can only have edges to vertices in V_1. Hence a minimum vertex cover of G is at most the size of $vc(G') + |V_1^x|$.
  - $vc(G) \geq vc(G') + |V_1^x|$
    This direction follows directly from Theorem3.9(Nemhauser-Trotter).

- $vc^*(G) = vc^*(G') + |V_1^x|$
  - $vc^*(G) \leq vc^*(G') + |V_1^x|$
    For any feasible solution y' of LPVC(G'), define a vector $\mathbf{y}$ as $y_v = y'_v$ for $v \in G'$ and $y_v = x_v$ for $v \in V(G) \setminus V(G')$. Then $\mathbf{y}$ is a feasible solution to LPVC(G) with value $vc^*(G') + |V_1^x|$.
  - $vc^*(G) \geq vc^*(G') + |V_1^x|$
    This direction is proven by noticing that $\mathbf{x}$ restricted to G' is a feasible solution of $vc^*(G')$ of size $vc^*(G) - |V_1^x|$.

□

As reduction rule 4 is just repeatedly executing reduction rule 3, exhaustively applying reduction rule 4 will also not affect the parameter.

Lemma 6.4. *Let (G', k') be the reduced instance after exhaustively applying reduction rule 4 on (G,k). Then $\mu(G', k') = \mu(G, k)$*

6.2 Branching

A branching algorithm for a parameterized problem is an algorithm that exhaustively searches through the solution space of the problem to find its optimal solution. A branching algorithm can be viewed as a rooted tree, where the root is the original instance from where the search begins and all the vertices in the tree are sub problems of the original instance. We call this tree the search tree of the algorithm. In each vertex v in the tree, the algorithm finds smaller subproblems on which it can be proven that there exist a solution to the instance corresponding to v, if and only if there exist a solution to at least one of the subproblems. The different subproblems are called branches in the search tree. A crucial part when finding sub problems on which the algorithm can branch is that these new problems must be smaller and provable closer to terminating the algorithm. As a branch in the algorithm terminates when the parameter is zero, this is measured in the drop of the parameter.

6.2.1 Time analysis

To analyze the running time of a branching algorithm one finds an upper bound to the number of leaves in the search tree. As each node in the tree has at least two children, this will also bound the number of nodes in the tree. Lets say an instance I can be split up into k sub problems $(I_1....,I_k)$ where in each subproblem I_l the parameter is reduced by i_l. If the parameter of I was μ, then we can bound the number of leaves for I by the recursion:

$$T(\mu) = T(\mu - i_1) + T(\mu - i_2) + ... + T(\mu - i_k)$$

In this thesis the branching rules will only simplify the instance into two branches where the parameter drops either by one half in both branches or by one half and one. The only two recursion we need to know is therefore:

$$\mathrm{T}(\mu) = \mathrm{T}(\mu - \tfrac{1}{2}) + \mathrm{T}(\mu - \tfrac{1}{2})$$

$$\mathrm{T}(\mu) = \mathrm{T}(\mu - \tfrac{1}{2}) + \mathrm{T}(\mu - 1)$$

which will give upper bounds on the number of leaves in the search tree by $\mathcal{O}(4^\mu)$ and $\mathcal{O}(2.6181^\mu)$ respectively.

6.2.2 Branching strategy

The branching phase of this algorithm follows a well known strategy for finding a vertex cover; for any vertex v in the graph, either v or all of its neighborhood has to be included in any valid vertex cover.

Lemma 6.5. *For a vertex $v \in V(G)$, in any valid vertex cover S for G, either v is in S or all of its neighbors must be in S.*

Proof. Assume S is a valid vertex cover for G and that there exists a vertex v such that neither v or all the vertices in N(v) are in S. Let y be one of the vertices in N(v) not in S. Then there will exist an edge, (v,y) $\in$ E(G) not covered by S, contradicting it as a valid vertex cover. Hence either v or all of N(v) must be in any valid vertex cover for G. □

Lemma 6.5 gives us the result needed for a branching strategy that exhaustively searches the solution space for the Vertex Cover above LP problem. Assume that none of the given reduction rules are applicable. Especially that reduction rule 4 is no longer applicable is important, as this guarantees that the instance (G,k) on which we branch have a surplus of at least one.

Branching Rule 1. *For a graph G, select an arbitrary vertex v and branch in two cases. In the first, include v in the vertex cover, remove v from the graph and work recursively on the instance $G' = G - v$ with $k' = k - 1$. In the other branch, include N(v) in the vertex cover, remove N[v] from the graph and work recursively on the instance $G' = G \setminus N[v]$ with $k' = k - |N(v)|$. This gives us the following recursion:*

$$VC(G,k) \text{ iff } VC(G-v,k-1) \text{ or } VC(G \setminus N[v], k - |N(v)|)$$

In order to analyze the running time of the algorithm we need to understand how each branch affects the parameter. We will analyze the two branches individually and we will see that in both branches the parameter is guaranteed to drop by at least one half. First we need to show how the branching rule affects the optimal LPVC(G) solutions:

Lemma 6.6. *Let G be a graph where $\boldsymbol{x} \equiv \frac{1}{2}$ is the unique optimal LPVC(G) solution. Then, for any vertex v in G, the value of an optimal solution to LPVC(G-v) is bounded by:*

$$vc^*(G-v) \geq vc^*(G) - \tfrac{1}{2}$$

Proof. Assume $vc^*(G-v) < vc^*(G) - \frac{1}{2}$. Let x' be an optimal solution to $vc^*(G-v)$, then x' extended with the variable $x_v = 1$ would be an optimal solution to LPVC(G) with integer values, contradicting the unique optimality of $\mathbf{x} \equiv \frac{1}{2}$. □

Lemma 6.7. *For a graph G where $\boldsymbol{x} \equiv \frac{1}{2}$ is the unique optimal LPVC(G) solution, the value of LPVC($G \setminus N[v]$) for a vertex v is bounded by:*

$$vc^*(G \setminus N[v]) \geq vc^*(G) - |N(v)| + \tfrac{1}{2}$$

Proof. Assume $vc^*(G \setminus N[v]) \leq vc^*(G) - |N(v)|$. Let x' be an optimal solution to LPVC($G \setminus N[v]$) and let x'' be an extension of x' where $x_v = 0$ and $x_u = 1$ for every u in N(v). Clearly this will be a feasible solution to LPVC(G) and $\mathbf{w}(x'') = \mathbf{w}(x') + |N(v)|$. If the claim $vc^*(G \setminus N[v]) \leq vc^*(G) - |N(v)|$ holds, this will be an optimal solution for LPVC(G) with integer values, contradicting the unique optimality of $\mathbf{x} \equiv \frac{1}{2}$. □

Lemma 6.8. *Branching rule 1 yields an $\mathcal{O}(4^\mu)$ upper bound on the number of leaves in the search tree.*

Proof. We prove the two branches separately:

- We first prove the case where a vertex v is included in the vertex cover. Let G and k be the instance on which we branch after exhaustively applying reduction rule 4. The new instance after including v in the vertex cover is then $G' = (G - v)$ with parameter $k' = k - 1$.

  From reduction rule 4 we know that $\mathbf{x} \equiv \frac{1}{2}$ was the unique optimal solution to LPVC(G). To understand how the parameter $\mu(G', k')$ relates to $\mu(G, k)$, we need a bound on $vc^*(G - v)$. Lemma 6.6 provides this bound and we can prove that the parameter $\mu(G, k)$ drops with at least one half in a branch where a vertex v is included in the vertex cover:

  $$\mu(G', k') = k' - vc^*(G') \leq k - 1 - (vc^*(G) - \tfrac{1}{2}) = \mu(G, k) - \tfrac{1}{2}$$

  Remark that the parameter in this branch drops by exactly one half as a bigger drop would imply that an optimal LPVC(G-v) solution together with x_v set to one would give an optimal LPVC(G) solution, contradicting the unique optimality of $\mathbf{x} \equiv \frac{1}{2}$ provided by reduction rule 4.

- The case where we include the neighborhood of v is very similar. Let $G' = G \setminus N[v]$ and $k' = k - |N(v)|$, again after exhaustively applying reduction rule 4 to obtain G and k. Lemma 6.7 provides the lower bound on $vc^*(G \setminus N[v])$ needed. We can now see that the parameter in a branch where the neighborhood of a vertex v is included in the vertex cover and the closed neighborhood of v is removed from the graph also drops by one half:

  $$\mu(G', k') = k' - vc^*(G \setminus N[v]) \leq k - |N(v)| - (vc^*(G) - |N(v)| + \tfrac{1}{2}) = \mu(G, k) - \tfrac{1}{2}$$

This proves that in each branch the parameter drops by at least one half, yielding the $\mathcal{O}(4^\mu)$ upper bound on the number of leaves in the search tree. □

6.3 Algorithm

Reduction Rule 4 and Branching Rule 1 gives us the theoretical tools needed for an FPT algorithm parameterized by $\mu(G, k)$ for Vertex Cover Above LP. In each iteration it reduces the graph as much as possible, until the reduced kernel consists of a graph where $\mathbf{x} \equiv \frac{1}{2}$ is the unique optimal solution. It then branches into two branches according to branching rule 1.

Theorem 6.9. *Vertex Cover above LP can be solved in* $\mathcal{O}(4^{\mu} \cdot mn^{\frac{3}{2}})$ *time.*

Proof. From lemma 6.8 we have bounded the number of leaves and therefore also the number of nodes in the search tree by $\mathcal{O}(4^{\mu})$. All the reduction rules performed on each node in the search tree can be performed in time $\mathcal{O}(mn^{\frac{3}{2}})$ by lemma 6.1. This gives us the claimed running time for the algorithm. The soundness of the algorithm follows from the soundness of all reduction rule and the fact that the branching strategy exhaustively searches through the solution space. □

7 Improved exponential algorithm

In this section we will present an algorithm that builds on the algorithm in the previous section and that, by adding some new reduction rules, improves the exponential running time to $\mathcal{O}^*(2.6181^{\mu})$. The algorithm presented was published in the paper "Faster Parameterized Algorithms using Linear Programming" by Loksthanov et.al. [8]. In this paper the algorithm was given as a step in explaining an algorithm with an even better worst-case dependence on μ, $\mathcal{O}^*(2.3146^{\mu})$. In this thesis we will only focus on the $\mathcal{O}^*(2.6181^{\mu})$ algorithm.

The algorithm in this section will further exploit the surplus of a graph. We will see that adding two simple reduction rules, both applicable in polynomial time, will guarantee that the graph has a surplus of at least two. With this surplus, the neighborhood branching strategy will yield a drop by at least one half in the parameter in the branch where a vertex v is included in the vertex cover and a drop by at least one in the branch where the neighborhood of v is included. This gives the $\mathcal{O}^*(2.6181^{\mu})$ upper bound on the number of leaves in the search tree.

7.1 Reduction

The first part of the reduction phase is to reduce the graph to an instance where the all-half vector is the unique optimal LPVC(G) solution. Again we use reduction rule 4 to obtain such an instance. In addition to this reduction rule we will introduce two new rules which handle all the surplus one sets in the graph. After all rules have been exhaustively applied, we are left with

an instance where the all-half solution is the unique optimal solution and the surplus of the instance is at least two.

Lemma 7.1. *Let G be a graph and Z an independent set such that* ***surplus****(Z′)* $\geq$ ***surplus****(Z) for all subsets* $Z' \subseteq Z$*. Then there exists a minimum vertex cover for G that either contains all of Z or none of Z.*

Proof. Let Z be an independent set with surplus i in G, satisfying the conditions of the lemma. Let S be a minimum vertex cover of G not satisfying the conditions in the lemma. In other words S does not contain all of Z nor all of N(Z). We will then show that $S \setminus Z \cup N(Z)$ will also be a minimum vertex cover for G.

Let $X = Z \setminus S$ and $Y = Z \cap S$. As $Z = X \cup Y$ has a surplus of i, we know that $|N(Z)| = |X| + |Y| + i$. As S is a valid vertex cover, all of N(X) must be in S. Given that X has a surplus of at least i, $|N(X)| \geq |X| + i$. This means there are at most $|Y|$ vertices in $N(Z) \setminus N(X)$. Removing Y from S and including all of N(Z) to S will clearly be a valid vertex cover for G. As we then have removed $|Y|$ vertices from S and added at most $|N(Z) \setminus N(X)| \leq |Y|$, $S \setminus Z \cup N(Z)$ will also be minimum and we have proven the lemma. □

We can state a more specific lemma which will help us prove some important results later:

Lemma 7.2. *Let G be a graph with* ***surplus****(G)* ≥ 1 *and let Z be an independent set with* ***surplus****(Z) = 1. Then there exist an optimal vertex cover S for G such that either all of Z and none of N(Z) is in S or none of Z and all of N(Z) is in S:*

1. $Z \subseteq S$ *and* $N(Z) \cap S = \emptyset$*, or*
2. $Z \cap S = \emptyset$ *and* $N(Z) \subseteq S$

Proof. According to lemma 7.1 there exist an optimal vertex cover such that either all of Z or none of Z is in S. We use this to prove the two cases:

- Let S be an optimal vertex cover such that $Z \subseteq S$, in other words, all of Z is in S. Then, if $N(Z) \cap S = \emptyset$, the conditions in 1 are already satisfied. If $N(Z) \cap S \neq \emptyset$, notice that then $S' = S \setminus Z \cup N(Z)$ will also be an optimal vertex cover, satisfying the conditions in 2.
- Let S be an optimal vertex cover such that $Z \cap S = \emptyset$. As S is a valid vertex cover this implies that all of N(Z) must be in S and thus proving the claim.

□

These lemmas gives us all we need to prove the two reduction rules in this chapter. We always assume that reduction rules of a lower number are no longer applicable before applying a reduction rule.

Reduction Rule 5. *For an instance (G,k), let Z be an independent set in G with surplus one such that the graph induced by N(Z) is not an independent set. Then we can put N(Z) in the suggested solution and create a new equivalent instance* $G' = G \setminus N[Z]$ *and* $k' = k - |N(Z)|$.

Lemma 7.3. *Reduction rule 5 is safe.*

Proof. As reduction rule 4 is no longer applicable, **surplus**(G) ≥ 1 and Z will be a set of minimum surplus, automatically satisfying the conditions of lemma 7.1. This means that there either exist a minimum vertex cover S for G containing none of Z or there exist a minimum vertex cover S' containing all of Z. If none of Z is in S then all of N(Z) must be for S to be a valid vertex cover. If all of Z is included in S' there will be at least one vertex v in $S' \cap N(Z)$ as N(Z) is not independent. In this case $S' = S \setminus Z \cup N(Z)$ will also be a minimum vertex cover of G. Hence we have shown that given the conditions of the lemma, there exist a minimum vertex cover for G containing all of N(Z). □

The following lemma shows how applying reduction rule 5 will affect the parameter $\mu(G, k)$:

Lemma 7.4. *Let* (G', k') *be the new reduced instance after applying reduction rule 5 on (G,k). Then* $\mu(G', k') \leq \mu(G.k) - \frac{1}{2}$

Proof. Given an instance (G,k), assume Z is an independent set found by reduction rule 5. Remember that the all-half solution was the unique optimal solution to LPVC(G). Let (G', k') be the new instance after the reduction rule has been applied. Then the weight $vc^*(G')$ of an optimal solution to LPVC(G') will be:

$$vc^*(G') = vc^*(G) \text{ - } \tfrac{1}{2}(|Z| + |N(Z)|) - \tfrac{1}{2}\left|V_0^{x'}\right| + \tfrac{1}{2}\left|V_1^{x'}\right|$$

As both Z and $V_0^{x'}$ are independent sets in G, the union of them will surely also be an independent set in G. We can easily see that $\mathrm{N}(Z \cup V_0^{x'}) = N(Z) \cup V_1^{x'}$. By adding and subtracting the size of N(Z) we obtain the following equation:

$$vc^*(G') = vc^*(G) \text{ - } |N(Z)| + \tfrac{1}{2}((|N(Z)| + \left|V_1^{x'}\right|) - (|Z| + \left|V_0^{x'}\right|))$$

With a minimum surplus of one and as $N(Z) \cup V_1^{x'}$ is the neighborhood of $Z \cup V_0^{x'}$, This gives us:

$$vc^*(G') \geq vc^*(G) \text{ - } |N(Z)| + \tfrac{1}{2}$$

With this bound on $vc^*(G')$ we can now bound the new parameter:

$$\mu(G', k') = k' - vc^*(G') \leq k - |N(Z)| - (vc^*(G) - |N(Z)| + \tfrac{1}{2}) = k - vc^*(G) - \tfrac{1}{2} = \mu(G, k) - \tfrac{1}{2}$$

□

Reduction Rule 6. *For an instance (G,k), let Z be an independent set with surplus one such that the graph induced by N(Z) is an independent set. Then we can create a new instance (G', k'), where G' is created by removing all of N[Z] from G and adding a new vertex z, making all vertices in $V(G) \setminus N[Z]$ that were adjacent to a vertex in N(Z) adjacent to z. Decrement k by the size of Z to obtain the new k'. Then (G,k) is a yes instance iff (G', k') is a yes instance.*

Lemma 7.5. *Reduction rule 6 is safe.*

Proof. We show both implications separately:

- $(G, k) \Rightarrow (G', k')$
  According to lemma 7.2, there exist an optimal vertex cover S of G such that either all of Z is in S or all of N(Z) is. We consider the two cases:
  - If $Z \subseteq S$ and $N(Z) \cap S = \emptyset$, all the edges from $N(Z)$ to $G \setminus N[Z]$ must be covered from $G \setminus N[Z]$. This means that $S' = S \setminus Z$ will cover all edges in G', including the edges to the new vertex z. $|S'| = k - |Z|$, proving this case.
  - If $Z \cap S = \emptyset$ and $N(Z) \subseteq S$, at least one of the edges between $N(Z)$ and $G \setminus N[Z]$ must be covered from $N(Z)$, otherwise $\overline{S} = S \setminus N(Z) \cup Z$ would be an vertex cover smaller than S, contradicting the optimality of S.
    Then $S' = S \setminus N(Z) + \{z\}$ will be a vertex cover for G' with size $k - |N(z)| + 1 = k - (|Z| + 1) + 1 = k - |Z|$
- $(G, k) \Leftarrow (G', k')$
  Let S' be a vertex cover for G' of size $k' = k - |Z|$. We again consider two cases, one for when z is in S' and for when z is not in S':
  - If z is in S', then $S = S' \setminus \{z\} \cup N(Z)$ will be a vertex cover of G with size $k' - 1 + |N(Z)| = k - |Z| - 1 + |Z| + 1 = k$
  - If z is not in S', then $S = S' \cup Z$ will be a vertex cover of G of size $k' + |Z| = k - |Z| + |Z| = k$.

□

Again, we need to analyze how applying the reduction rule will affect the parameter:

Lemma 7.6. *Let (G', k') be the new reduced instance after applying reduction rule 6 on (G,k). Then $\mu(G', k') \leq \mu(G.k)$*

Proof. If we can prove the following claim then the upper bound on the new parameter follows directly.

Claim 1. *The weight $vc^*(G')$ of an optimal LPVC(G') solution is bounded by:*

$$vc^*(G') \geq vc^*(G) \text{ - } |Z|$$

Proof. In order to prove the claim we chase a contradiction, assuming that $vc^*(G') \leq vc^*(G) \text{ - } |Z| - \frac{1}{2}$. This inequality can be re-written to $vc^*(G') + |Z| \leq vc^*(G) \text{ - } \frac{1}{2}$. We need to look at the three cases where x' is an optimal solution to LPVC(G') and the variable of the new vertex z, created by the reduction rule, admits the three different values in $\{0, \frac{1}{2}, 1\}$

- Case 1: $x'_z = 1$
  Create a new feasible LPVC(G) solution as follows:

  $$x''_v = \begin{cases} x'_v & v \in V(G') - \{z\} \\ 0 & v \in Z \\ 1 & v \in N(Z) \end{cases}$$

  The weight of this solution, given that our assumption holds, will be:

  $$\mathbf{w}(x'') = \mathbf{w}(x') \text{ - } 1 + |N(Z)| = \mathbf{w}(x') + |Z| \leq \mathbf{w}(\text{x}) \text{ - } \frac{1}{2}.$$

  which is a contradiction to the optimality of x.

- Case 2: $x'_z = 0$
  Create a new feasible LPVC(G) solution as follows:

  $$x''_v = \begin{cases} x'_v & v \in V(G') - \{z\} \\ 1 & v \in Z \\ 0 & v \in N(Z) \end{cases}$$

  The weight of this solution, given that our assumption holds, will be:

  $$\mathbf{w}(x'') = \mathbf{w}(x') + |Z| \leq \mathbf{w}(\text{x}) \text{ - } \frac{1}{2}.$$

  which is a contradiction to the optimality of x.

- Case 3: $x'_z = \frac{1}{2}$
  Create a new feasible LPVC(G) solution as follows:

$$x''_v = \begin{cases} x'_v & v \in V(G') - \{z\} \\ \frac{1}{2} & v \in Z \\ \frac{1}{2} & v \in N(Z) \end{cases}$$

  The weight of this solution, given that our assumption holds, will be:

$$\mathbf{w}(x'') = \mathbf{w}(x') \text{ - } \tfrac{1}{2} + \tfrac{1}{2}(|Z| + |N(Z)|) = \mathbf{w}(x') \text{ - } \tfrac{1}{2} + \tfrac{1}{2}(|Z| + |Z| + 1)$$
$$= \mathbf{w}(x') + |Z| \leq \mathbf{w}(\text{x}) \text{ - } \tfrac{1}{2}.$$

  which is a contradiction to the optimality of x.

□

Thus we have seen that $vc^*(G') \geq vc^*(G)$ - $|Z|$. We can use this to bound the new parameter:

$$\mu(G', k') = k' - vc^*(G') \leq k - |Z| - (vc^*(G) - |Z|) = k - vc^*(G) = \mu(G, k)$$

□

In order to apply the reduction rules we need an efficient way to find independent sets with surplus one in G. The following claim shows us how:

Claim 2. *For a graph G where reduction rule 4 is no longer applicable, there exists an independent set Z with* ***surplus****(Z) = 1 if and only if there exists a vertex v in G such that the optimal LPVC(G) value with x_v fixed to zero increases with exactly one half.*

Proof. Suppose there exist a vertex u in G such that the weight of an optimal LPVC(G) solution x' after fixing x'_u to zero is $\mathbf{w}(x') = \mathbf{w}(\text{x}) + \frac{1}{2}$. Let $Z = V_0^{x'}$ and $N(Z) = V_1^{x'}$. Then:

$$\mathbf{w}(x') = \mathbf{w}(\text{x}) \text{ - } \tfrac{1}{2}|Z| + \tfrac{1}{2}|N(Z)| = \mathbf{w}(\text{x}) \text{ - } \tfrac{1}{2}$$
$$\Rightarrow$$
$$|N(Z)| - |Z| = \mathbf{surplus}(\text{Z}) = 1$$

For the other direction, assume there exists an independent set Z with surplus one. Now create a new feasible LPVC(G) solution x' where all vertices in Z are assigned the value zero, all vertices in N(Z) are assigned one and the rest of the vertices are assigned one half. The weight of this new LPVC(G) solution must then be:

$$\mathbf{w}(x') = |N(Z)| + \tfrac{1}{2}(V(G) \setminus N[Z])$$

Now subtract the weight of x on both sides, which we know was the unique all half optimal solution to LPVC(G), and get:

$$\mathbf{w}(x') \text{ - } \mathbf{w}(\text{x}) = |N(Z)| - \tfrac{1}{2}(|Z| + |N(Z)|) = \tfrac{1}{2}(|N(Z)| - |Z|) = \tfrac{1}{2}$$

. □

Notice that the independent sets found in claim 2 will be inclusion-minimal, as the surplus of G before invoking the algorithm was at least one.

Lemma 7.7. *Reduction rules 5 and 6 can be found in* $\mathcal{O}(n(mn^{\frac{1}{2}})$ *time.*

Proof. Invoke the algorithm from claim 2. If it returns an independent set Z then:

- If Z is not an independent set, invoke reduction rule 5
- If Z is an independent set, invoke reduction rule 6

The algorithm from claim 2 will in worst case recompute the LPVC(G) value n times. To check whether the neighborhood of Z is independent can be done in linear time. This gives the total $\mathcal{O}(n(mn^{\frac{1}{2}})$ running time.

□

7.2 Branching

The branching part of this algorithm follows the same strategy as the previous algorithm. After all reduction rules have been exhaustively applied, pick a vertex v and branch on either including v or all of its neighborhood in the vertex cover. This branching rule was proven to be correct in section 6.2, but we still need to analyze how the parameter drops now that two more reduction rules have been added.

Lemma 7.8. *After exhaustively applying all the reduction rules, branching according to branching rule 1 will yield an upper bound on the size of the search tree by* $\mathcal{O}^*(2.618^{\mu(G,k)})$

Proof. The two branches are proved separately:

- The case where the vertex v is included in the vertex cover is equal to the first part of lemma 6.8, which proves that the parameter drops by at least one half in the new instance (G', k'):

$$\mu(G', k') \leq \mu(G, k) - \tfrac{1}{2}$$

- Let (G', k') be the new instance created from (G,k) after including the neighborhood of v in the vertex cover and removing N[v] from G. Then $k' = k - |N(Z)|$. We know that $vc^*(G) = \frac{|V(G)|}{2}$ as reduction rule 4 was no longer applicable on G and that **surplus**(G) ≥ 2 as reduction rules 5 and 6 were no longer applicable. To prove the claimed drop of the parameter, it is sufficient to prove a lower bound to LPVC(G'):

$$vc^*(G \setminus N[v]) \geq vc^*(G) - |N(v)| + 1 \tag{4}$$

To prove this, we assume the contrary:

$$vc^*(G \setminus N[v]) \leq vc^*(G) - |N(v)| + \frac{1}{2} \tag{5}$$

Let **x'** be an optimal solution to LPVC($G \setminus N[Z]$) and let **x** be the extension of **x'** where $x_v = 0$, $x_u = 1$ for all $u \in N(V)$ and the variables for vertices in $G \setminus N[v]$ stays the same. **x** will clearly be feasible and its weight will be:

$$\mathbf{w(x)} = \mathbf{w(x')} + |N(v)| \leq vc^*(G) - |N(v)| + \tfrac{1}{2} + |N(v)| = \tfrac{|V(G)|}{2} + \tfrac{1}{2}$$

But now lemma 5.3 tells us that:

$$\mathbf{surplus}(V_0^x) = 2 \cdot \mathbf{w(x)} - |V(G)| \leq 2 \cdot (\tfrac{|V(G)|}{2} + \tfrac{1}{2}) - |V(G)| = 1$$

This contradicts that **surplus**(G) ≥ 2, proving the lower bound on $vc^*(G')$. With this lower bound the parameter of the new instance will drop by at least one:

$$\mu(G', k') = k' - vc^*(G') \leq k - |N(v)| - (vc^*(G) - |N(v)| + 1) = k - vc^*(G) - 1 = \mu(G, k) - 1$$

□

7.3 Algorithm

We have seen that exhaustively applying reduction rules 4, 5 and 6 will give a reduced kernel instance (G', k') where the all-half solution is the unique optimal solution to LPVC(G') and **surplus**(G') ≥ 2. Lemma 7.7 proved that the reductions 5 and 6 can be found in $\mathcal{O}(mn^{\frac{3}{2}})$ time. Using the neighborhood branching strategy lemma 7.8 showed that the search tree will have an upper bound of $\mathcal{O}(2.6181^{\mu})$ leaves. All these results gives us the main theorem of this section:

Theorem 7.9. *Vertex Cover above LP can be solved in* $\mathcal{O}(2.6181^{\mu} \cdot mn^{\frac{3}{2}})$ *time.*

8 Linear Time FPT Algorithm

Up until this point the algorithms presented have found the LPVC(G) solutions by reducing it to finding a vertex cover in a constructed bipartite graph. This has yielded a polynomial running time as the algorithm used for finding vertex cover in bipartite graphs, Hopcroft-Karp, runs in time $\mathcal{O}(m\sqrt{n})$. In this chapter we will introduce and explain an algorithm that reduces the polynomial running time so it becomes linear. The algorithm is from the paper "Linear-Time FPT Algorithms via Network Flow "[6] by Yoichi Iwata, Keigo Oka and Yuichi Yoshida. In this paper the authors consider weighted graphs but, as we are only concerned with finding vertex cover in unweighted graphs, the results from the paper are restated and simplified for the unweighted case.

The algorithm from [6] proceeds much like the algorithm presented in section 6. The main difference is the way an optimal LPVC(G) solution is computed. In this algorithm a maximum flow in a flow network is computed and an optimal solution to LPVC(G) is found from properties of its residual graph. We will see later that this is really similar to what we have already done as there are a lot of similarities between the network and the bipartite graph(definition 3.6) created from G.

A key observation that allows this algorithm to achieve a linear running time is that the flow in the network can be re-used after applying a reduction rule. With re-use we mean that all edges with flow not incident to a vertex that has been removed keep its flow. In fact, the flow restricted to the new reduced instance is still optimal after reducing the network according to reduction rule 3. Hence the algorithm avoids time consuming re-computations and the only time the flow has to be re-computed is after branching. But, as we will see, this flow update will only need at most one breadth-first search in each branch, making the running time linear.

8.1 Reduction

As in the previous algorithms this algorithm will also find sets V_0, $V_{\frac{1}{2}}$ and V_1 based on an optimal LPVC(G) solution and reduce the graph accordingly. Stated in surplus terminology, it first finds all the inclusion minimal independent sets with non-positive surplus and removes them according to reduction rule 3. Then, all surplus zero sets in the network are found and also removed according to reduction rule 3. In the previous algorithms the surplus zero sets have been found by completely re-computing LPVC(G) several times. It is easy to observe that these re-computations do a lot of unnecessary work, as the restrictions of LPVC(G) solutions by removing one or some variables are never far from optimal. The algorithm in this section exploits connectivity properties in the residual graph of a network with a maximum flow, completely avoiding to re-compute LPVC(G) values, to find

the surplus zero sets.

8.1.1 Flow networks

A flow network is a directed graph with weighted edges. The intuition is that each edge is a pipe with a capacity in which a flow can be pushed through. The capacity of an edge is represented by its weight. A flow network has two special vertices. A source s in which all flow originates and a sink t in which all flow terminates. One can think of the source as a vertex that can push infinite amount of flow to the network.

We can formally define a flow network as:

Definition 8.1. *$N = (V \cup \{s, t\}, E)$ with a capacity function $c : E \to \mathbb{R}^+$*

A flow in a network is a real valued function assigning a flow to the edges in a network:

Definition 8.2. *A flow $f : E \to \mathbb{R}$ is valid for a network N if:*

- $f(u, v) \leq c(u, v)$, $\forall (u, v) \in E$
- $f(u, v) = -f(v, u)$, $\forall (u, v) \in E$
- $\sum_{w \in V} f(u, w) = 0$, $\forall u \in V \setminus \{s, t\}$

Given a valid flow f in a network N, the amount of flow running through a vertex v equals $\sum_{e \in \delta(v)^+} f(e)$. The amount of flow in the network is the total amount of flow running in to t, $\sum_{e \in \delta(v)^-} f(e)$, denoted $val(f)$. Due to flow conservation, the third condition for a valid flow, this amount has to equal the sum of flow running out from s. When we have a flow in a network the graph induced by the amount of capacity left in each edge is called the residual graph of N:

Definition 8.3. *For a network N with a flow f, define G_f to be the residual graph where the vertices in G_f are identical to the vertices in N and where for every edge (u,v) in N then:*

1. *If $f(u, v) < c(u, v)$, create a forward edge (u,v) in G_f with capacity $c_f(u, v) = c(u, v) - f(u, v)$*
2. *If $f(u, v) > 0$, create a backward edge (v,u) in G_f with capacity $c_f(v, u) = f(u, v)$*

For a network N and a flow f, whenever the flow of an edge has saturated its capacity it will not be present in the residual graph. The residual graph thus is a representation of available flow in the network. When finding a maximum flow in a network, augmenting paths in the residual graph are of

vital importance. An augmenting path in a residual graph is a path from s to t. As all edges in the residual graph have a positive residual capacity this means an augmenting path is a way to push more flow from s to t. The capacity of an augmenting path is the minimum residual capacity of any edge in the path. Thus, when finding an augmenting path, the flow in the network can be augmented by increasing the flow running through every edge of the path by the paths capacity. Analogous to maximum matchings in bipartite graphs, a flow f in a network is maximum iff there exist no augmenting path in the residual graph G_f. This fact is used in the Ford-Fulkerson algorithm for finding maximum flows in flow network, which exhaustively searches for augmenting paths and augment the flow accordingly.

Connectivity in the residual graph of a network is important for the algorithm to achieve a linear running time.

8.1.2 Primal and dual LP

In order to efficiently compute solutions to our primal linear program, this algorithm keeps track of a dual linear program. It is well known from LP duality that every primal LP has a dual LP that admits the same optimal value. We will later show that we from an optimal dual solution indeed can efficiently compute an optimal primal solution. We refer the reader to "Linear Programming: Foundations and Extensions" by Vanderbei [11] for an introduction to linear programming and duality. The only properties of LP duality that we will use in this thesis are captured in lemma 8.2. A formalization of LP duality is therefore not necessary in this text and we will only state the few properties of duality that we need. First we will formally define the dual:

Definition 8.4. *Dual Linear Program for Vertex Cover*

$$
\begin{aligned}
&\textit{maximize:} && \sum_{e \in E(G)} y_e && \\
&\textit{subject to:} && \sum_{e \in \delta(v)} y_e \leq 1 && \forall v \in V(G) \\
& && y_e \geq 0 && \forall e \in E(G) \\
& && y_e \in \mathbb{R} && \forall e \in E(G)
\end{aligned}
$$

We can show that the value of any feasible solution to the dual LP will be smaller or equal to the value of any feasible solution to the primal LPVC(G):

Lemma 8.1. *Given any feasible solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ to the primal and dual LP respectively, then:*

$$\boldsymbol{w(y) \leq w(x)}$$

Proof. Let $\mathbf{x}$ and $\mathbf{y}$ be any feasible primal and dual solutions. Then:

$$\begin{aligned}
\mathbf{w(x)} &= \sum_{v \in V(G)} x_v \\
&= \sum_{v \in V(G)} x_v \cdot 1 \\
&\geq \sum_{v \in V(G)} x_v \sum_{e \in \delta(v)} y_e \\
&= \sum_{v \in V(G)} \sum_{e \in \delta(v)} x_v y_e \\
&= \sum_{e \in E(G)} \sum_{v \in (e)} y_e x_v \\
&= \sum_{e \in E(G)} y_e \sum_{v \in e} x_v \\
&\geq \sum_{e \in E(G)} y_e \cdot 1 = \mathbf{w(y)}
\end{aligned} \tag{6}$$

The first inequality follows from the constraints of the dual LP, that all edges incident to a vertex v can have a total value of at most one. Then, notice that taking the sum over all vertices v in G and then the sum over all edges incident to v will be the same as taking the sum over all edges e in G and then the sum over vertices adjacent to e. The last inequality follows from the constraints of the primal LP, the value of the vertices adjacent to an edge e must be at least one.

□

The following lemma is a direct consequence of the strong duality theorem of linear programming [11] that states that the optimal solutions to an LP and its dual have the same value.

Lemma 8.2. *For optimal solutions* $\boldsymbol{x^*}$ *and* $\boldsymbol{y^*}$ *to the primal and dual LP respectively, then:*

$$\boldsymbol{w(x^*) = w(y^*)}$$

The solutions to the dual LPVC(G) can be represented as a maximum flow in a flow network constructed from G. To avoid confusion, we will from now on use an overline whenever we are talking about flow networks. A flow network created from a graph G will be denoted $\overline{G}$ and also any subsets of vertices or edges in the network will have an overline.

Definition 8.5. *From a graph G we construct the network* $\overline{G} = (\overline{V} \cup \{s, t\}, \overline{E}, c)$ *with capacity function* $c : \overline{E} \to \mathbb{N}$ *as follows:*

- $\overline{V} = \overline{L} \cup \overline{R}$, *where* $\overline{L} = \{l_v : v \in V(G)\}$ $\overline{R} = \{r_v : v \in V(G)\}$
- $\overline{E} = \{(s, l_v) : l_v \in \overline{L}\} \cup \{(r_v, t) : r_v \in \overline{R}\} \cup \{(l_u, r_v), (l_v, r_u) : (u, v) \in E(G)\}$
- $$c(e) = \begin{cases} 1 & \forall e \in \{(s, l_v) : l_v \in \overline{L}\} \cup \{(r_v, t) : r_v \in \overline{R}\} \\ \infty & otherwise \end{cases}$$

For a subset $\overline{S} \subseteq \overline{V}$, define $S_L = \{v \in V(G) : l_v \in \overline{S}\}$ and $S_R = \{v \in V(G) : r_v \in \overline{S}\}$

8.1.3 Relations between matchings in H_G and flow in $\overline{G}$

To show the similarities between the two approaches computing optimal primal LPVC(G) solutions we first remark that the network induced by $\overline{V}$ is just a directed version of the bipartite graph H_G in definition 3.6. We want to show that there is a one to one correspondence between finding minimum vertex covers in H_G and minimum vertex covers in the network induced by $\overline{V}$. This will later be used to prove how to compute a primal LPVC(G) solution from a maximum flow in $\overline{G}$.

The first step is to prove that there exists an optimal flow for $\overline{G}$ with only integer values:

Lemma 8.3. *A maximum flow f* in $\overline{G}$ found by the Ford-Fulkerson algorithm will only admit integer values.*

Proof. We can prove this by an inductive argument. Start with an empty flow f. The first augmenting path P found when looking for a maximum flow will have capacity one, as all edges from the source and to the sink have capacity one and the rest of the edges start with infinite capacity. Now, in the residue graph created by the updated flow, the edges in P from the source and to the sink will not be present, as their capacity has been saturated. Let (u_l, v_r) be the edge in which we now have a flow. The new back edge (v_r, u_L) created in the residual graph will then have a capacity of one.

Now, when looking for a new augmenting path, the minimum capacity of an edge in the network will be one. The new augmenting path found will therefore have capacity one. This process will be repeated for every augmenting path found, always guaranteeing that they will have a capacity of one. Thus, when a maximum flow has been found it must admit only integer values. □

Corollary 8.4. *$\overline{G}$ has a maximum integer valued flow*

From this point, when talking about maximum flows in $\overline{G}$ we will always be talking about an integer flow. We can now show that there is a one to one correspondence between matchings in H_G and flows in $\overline{G}$

Lemma 8.5. *Let f be an integer valued flow in $\overline{G}$. Then $M = \{(u, v) : f^*(u, v) = 1, (u, v) \in \overline{E}(\overline{V})\}$ will be a matching in H_G with $|M| = val(f)$.*

Proof. All vertices in $\overline{L}$ and $\overline{R}$ can maximum have a flow of one running through them, which means that no two edges with a flow of one can share an end-point. As the graph induced by $\overline{V}$ corresponds to H_G we have proven the statement. The size follows from the construction. □

Lemma 8.6. *Let M be a matching in H_G, then the function $f : \overline{E} \to \mathbb{R}$ defined as:*

$$f(u,v) = \begin{cases} 1 & if (u,v) \in M \\ 1 & \text{if } u = s \text{ and } v \text{ endpoint of an edge in } M \\ 1 & \text{if } v = t \text{ and } u \text{ endpoint of an edge in } M \\ 0 & otherwise \end{cases}$$

will be a flow in $\overline{G}$ where $val(f) = |M|$.

Proof. To show that f is a flow we need to show that all conditions for a valid flow are satisfied. Clearly no edges in the network will be given a flow larger than its capacity. All the flow in to a vertex must also equal the flow out of the vertex as only one edge out of a vertex in $\overline{L}$ can have flow equal to one. Again the size follows from the construction. □

Corollary 8.7. *There is a one to one correspondence between maximum flows in $\overline{G}$ and maximum matchings in H_G*

Lemma 8.8. *Given a a graph G and a maximum flow f* in $\overline{G}$. Let Z be the set of vertices reachable from s in G_{f^*}. Then,*

$$S = (\overline{L} \setminus Z) \cup (\overline{R} \cap Z)$$

will be an optimal vertex cover for H_G where $|S| = val(f^*)$.

Proof. Let M be the maximum matching in H_G corresponding to f*. We will show that the vertices in Z are the same as all vertices reachable by an alternating path relative to M, from the free vertices in L_H. Recall that free vertices in L_H are the vertices not incident to an edge in the matching. Using lemma 3.15 this will prove the claim.

All the edges going out from s in G_{f^*} must have a capacity of one and thus no flow running through them. The vertices in $\overline{L}$ with an edge from s in G_{f^*} will therefore correspond to the free vertices in L_H, as no edges out from these vertices can have a flow and thus be in the matching M.

In G_{f^*}, the only edges going from $\overline{R}$ to $\overline{L}$ are edges in which the edge going in the opposite direction has a flow. Thus vertices in Z are the exact same as the vertices reachable be an alternating path from the free vertices in L_H.

To prove that $|S| = val(f*)$, notice that no flow can go from a vertex v in $\overline{L} \setminus Z$ to a vertex u in $\overline{R} \cap Z$, as then the edge (u,v) would be in the residue graph G_{f^*} and v would then be reachable from s and thus in Z. All the flow running through vertices in $\overline{L} \setminus Z$ must therefore go to $\overline{R} \setminus Z$ before it ends in t. This means that:

$$|S| = \left|\overline{L} \setminus Z\right| + \left|\overline{R} \cap Z\right| = val(f^*)$$

Corollary 8.7 proves that the size of a maximum matching M in H_G will be equal to $val(f^*)$. As any minimum vertex cover in a graph must be at least as big as a maximum matching, this proves the optimality of S.

□

8.1.4 Dual LPVC(G) as flow in $\overline{G}$

In the constructed network $\overline{G}$ we will create a flow from an optimal dual solution. In fact, as we will show, there exists a one to one correspondence between a feasible dual LP solution in G and a flow in the network $\overline{G}$. It is due to this correspondence that finding primal LP solutions in G is reduced to finding subsets in $\overline{G}$ with certain properties. We will first show how to create a flow from a feasible dual LP solution. Remember that $\delta(v)$ denotes all the edges incident to v.

Lemma 8.9. *Given a dual solution* $\boldsymbol{y}$ *to a graph G, define* $f : \overline{E} \to \mathbb{R}$ *as:*

$$f(e) = \begin{cases} \sum_{e \in \delta(v)} y_e & for\ (s, l_v)\ and\ (r_v, t) \in \overline{E} \\ y_e & otherwise \end{cases}$$

Then f is a flow in $\overline{G}$ *of amount* $2 \cdot \boldsymbol{w(y)}$*, i.e.* $val(f) = 2 \cdot \boldsymbol{w(y)}$.

Proof. In order to prove that f is a valid flow we need to show that no edge is given a flow more than its capacity and that the amount of in-flow to each vertex in the network, except s and t, is equal to the amount of out-flow from the vertex. The first condition follows directly from the fact that **y** is a feasible dual solution. The constraints of the dual says that the sum of all the edges incident to a vertex v can not be more than one. Hence none of the edges between the source and a vertex in $\overline{L}$ or between a vertex in $\overline{R}$ and the sink can be given a flow more than one, which is their capacity. All other edges have infinite capacity and hence will not be a problem.

It is easy to see that the amount of in-flow equals the amount of out-flow for every vertex in $\overline{V}$. The out-edges of each vertex $l_v \in \overline{L}$ are exactly the same as the edges incident to $v \in G$ and as the edges between $\overline{L}$ and

$\overline{R}$ have been given a flow equal to the dual-variable of the same edge it follows that the in-flow equals the out-flow. Each edge in E(G) will create two edges in $\overline{E}$, so the amount of flow is two times the value of $\mathbf{y}$. Hence, $val(f) = 2 \cdot \mathbf{w}(\mathbf{y})$, as claimed. □

Thus we have shown how to create a flow in $\overline{G}$ from a feasible dual LPVC(G) solution. We will now show the other direction, how to create a feasible dual solution to LPVC(G) from a flow in $\overline{G}$:

Lemma 8.10. *Given a flow f in $\overline{G}$, define the variables of the vector $\boldsymbol{y}$ as:*

$$y_e = \tfrac{1}{2}(f(l_u, r_v) + f(l_v, r_u)) \quad \textit{for all } e = (u, v) \in E(G)$$

Then $\boldsymbol{y}$ is a feasible dual LPVC(G) solution with weight half of the amount of flow in f.

Proof. We must show that $\sum_{e \in \delta(v)} y_e \leq 1$ for all $v \in V(G)$. As f is a valid flow and the capasity of edges between s and vertices in $\overline{L}$ are one, we know that the in-flow of a vertex $l_u \in \overline{L}$ is at most one and hence the out-flow is at most one:

$$\sum_{e=(l_u, r_v) \in \delta^+(l_u)} f(l_u, r_v) \leq 1$$

Similarly, edges between vertices in $\overline{R}$ and t have capacity one and hence the out-flow of a vertex $r_v \in \overline{R}$ can be at most one. This means that the in-flow is also restricted to one:

$$\sum_{e=(l_u, r_v) \in \delta^-(r_v)} f(l_u, r_v) \leq 1$$

The value of all edges incident to a vertex v in G is equal to half the flow going out of the left copy of v in the network plus half of the value flowing in to the right copy of v in the network:

$$\sum_{(u,v) \in \delta(v)} y_e = \tfrac{1}{2} \sum_{(l_v, r_z) \in \delta^+(l_v)} f(l_v, r_z) + \tfrac{1}{2} \sum_{(l_z, r_v) \in \delta^-(r_v)} f(l_z, r_v) \leq 1$$

The value $\mathbf{w}(\mathbf{y}) = \sum_{e \in E(G)} y_e = \frac{1}{2}(f(l_u, r_v) + f(l_v, r_u))$ is clearly half of the flow in the network. □

The two lemmas just presented give rise to the following corollaries:

Corollary 8.11. *From a maximum flow f^* in $\overline{G}$ we can compute an optimal solution $y^*$to the dual LP for G and vice versa.*

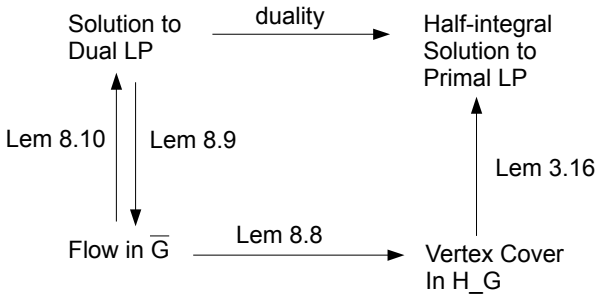
Corollary 8.12. *Given a graph G in which $\boldsymbol{x} \equiv \frac{1}{2}$ is an optimal primal LP solution, then the network $\overline{G}$ will have a maximum flow which saturates all its vertices, i.e. it will have a flow of amount one running through all vertices in $\overline{V}$.*

Proof. From lemma 8.2 we know that there exist an optimal dual solution $\mathbf{y^*}$ with $\mathbf{w(y^*)} = \mathbf{w(x)} = \frac{|V(G)|}{2}$. Lemma 8.11 then shows how to create a flow f* in $\overline{G}$ of amount $2 \cdot \mathbf{w(y^*)} = |V(G)|$. This means that all edges from the source to vertices in $\overline{L}$ must be saturated and hence we have a maximum flow in $\overline{G}$ □

Corollary 8.13. *The maximum amount of flow a network $\overline{G}$ can have is equal to $|V(G)|$.*

Proof. The maximum obtainable amount of flow in a network is obtained when all edges out from the sink have a flow of maximum capacity. In the network $\overline{G}$ there is one edge going from the source s to each of the vertices in $\overline{L}$, each with a capacity 1. Hence, no flow in $\overline{G}$ can exceed $\left|\overline{L}\right| = |V(G)|$. □

The following figure sums up some of the relations we have just proven:



8.2 Algorithm

As in the previous algorithms, this algorithm consists of a reduction part and a branching part. The reduction part follows a familiar strategy. Find independent sets with non-positive surplus, fix the variables accordingly and remove the vertices whose variables have integer values from the graph. Then it follows an even simpler branching strategy than before. For each edge (u,v) in the graph, either u or v has to be in the vertex cover to cover this edge, so we branch on including either of its end-points in the vertex cover, remove the end-point from G and reduce the parameter.

Where this algorithm differs from the others is that it finds the non-negative independent surplus sets via a flow in a flow network and then updates the flow in linear time. The algorithm is linear when an optimal dual solution is given as input. In other words, an optimal flow has to be found in the network as a pre-processing step before the algorithm starts. The algorithm was developed to find linear FPT algorithms for a family of NP-Hard problems, among them ALMOST-2-SAT and ODD CYCLE TRANSVERSAL, and in the reduction from these problems to Vertex Cover above LP

the optimal dual-solution is produced in linear time.

When using a branching strategy to exhaustively search for a solution to a problem a search tree will implicitly be created. On each node in this search tree the algorithm is given an instance, a parameter and an optimal dual solution represented as a flow in the network created from the instance. If the given instance has no edges and a positive parameter, we know that a solution has been found. If the parameter is negative, no solution can be found in this branch in the search tree. After checking these conditions, the algorithm proceeds with the following steps:

1. From an optimal dual solution, represented as a flow in the network, compute an optimal primal LP of G. Reduce G according to reduction rule 3.

2. Find all surplus zero sets in G and reduce G according to reduction rule 3, by finding subsets $\overline{S} \subseteq G_{f*}$ that corresponds to surplus zero sets in G. After this step G has surplus of at least one and the all-half vector as a unique optimal primal solution.

3. Branch on an arbitrarily selected edge e. In each branch, include one of the endpoints of e to be in the suggested solution, remove the endpoint v from G, update the flow in $\overline{G}$ and work recursively on the new instance (G-v, k-1).

We will describe the different steps separately in the following sections. Step one and two are the reduction part of the algorithm while step three is the branching part.

8.2.1 Step 1

We saw in section 8.1.3 how closely related the networks $\overline{G}$ and bipartite graphs H_G created from G are. In this step we will find an optimal primal LPVC(G) solution from the network $\overline{G}$ and then reduce the instance accordingly. As we already have a lot of results from previous sections on how to compute optimal primal solutions to LPVC(G) from H_G, we want to re-use these here. The motivation behind this is both to avoid stating a lot of results again and to see how the two approaches are connected.

Step one can be implemented by finding all vertices reachable from the source vertex s in the residual graph G_{f^*}:

Lemma 8.14. *Given a a graph G and a maximum flow f* in $\overline{G}$. Let Z be the set of vertices reachable from s in G_{f^*}. Then the vector $\boldsymbol{x}$ in $\mathbb{R}^{|V(G)|}$ with variables:*

$$x_v = \begin{cases} 0 & l_v \in Z, r_v \notin Z \\ 1 & l_v \notin Z, r_v \in Z \\ \frac{1}{2} & otherwise \end{cases}$$

will be an optimal solution to LPVC(G).

Proof. Lemma 8.8 shows that we can create an optimal vertex cover $S = (\overline{L} \setminus Z) \cup (\overline{R} \cap Z)$ for H_G from Z. Now the statement $l_v \in Z, r_v \notin Z$ is the same as saying that for a vertex v in V(G), none of l_v or r_v are in the vertex cover S and the statement $l_v \notin Z, r_v \in Z$ is the same as saying that for a vertex v in V(G) both l_v and r_v are in S. Thus the vector $\mathbf{x}$ is the exact same vector as found by lemma 3.16, proving the correctness of this lemma. The vertices in Z can be found by a single BFS search in G_{f*} and thus finding an optimal primal LPVC(G) solution can be done in linear time. □

After computing an optimal primal solution $\mathbf{x}$ to LPVC(G) from $\overline{G}_{f*}$, the vertices of G can be partitioned into the vertex sets V_0^x, V_1^x and $V_{\frac{1}{2}}^x$. Then reduction rule 3 allow us to remove the vertices in both V_0^x and V_1^x from the graph and its corresponding network and include all vertices in V_1^x to the suggested solution.

Let $\overline{G'}$ be the new reduced network after removing vertices in V_0^x and V_1^x and f' be the flow from f* restricted to $\overline{G'}$. In order to achieve a linear running time we want to avoid computing the maximum flow in $\overline{G'}$ from scratch, this turns out to be easy as f' can be proven to be optimal:

Lemma 8.15. *For a graph G, let*

- $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *be optimal primal and dual LP solutions for G*
- $\boldsymbol{x}$**'** *and* $\boldsymbol{y}$**'** *be the vectors where variables according to vertices in* $V_0^x \cup V_1^x$ *have been removed from* $\boldsymbol{x}$ *and* $\boldsymbol{y}$

Then $\boldsymbol{x}$**'** *and* $\boldsymbol{y}$**'** *will be optimal primal and dual solutions to* $G[V_{\frac{1}{2}}^x]$

Proof. **x'** will be an optimal primal solution to LPVC(G) due to Theorem 3.9(Nemhauser-Trotter). The weight of $\mathbf{x}$ is:

$$\mathbf{w}(\mathbf{x}) = \mathbf{w}(\mathbf{x}') + |V_1|$$

When the optimal dual solution $\mathbf{y}$ is restricted to $\mathbf{y}'$, all the variables of edges incident to V_1^x are removed from $\mathbf{y}$. V_0^x is an independent set in G, so the edges incident to V_1^x are the only edges removed. As $\mathbf{y}$ is a feasible dual solution to LPVC(G), we know that the values of edges incident to a vertex can not be more than one. The weight of $\mathbf{y}$ will therefore be:

$$\mathbf{w}(\mathbf{y}) = \mathbf{w}(\mathbf{y}') + \textstyle\sum_{e \in \delta(V_1)} y_e \leq \mathbf{w}(\mathbf{y}') + |V_1|$$

We then have:

$$\begin{aligned} \mathbf{w}(\mathbf{x}) &= \mathbf{w}(\mathbf{y}) \\ \mathbf{w}(\mathbf{x`}) + |V_1| = \mathbf{w}(\mathbf{y}) &\leq \mathbf{w}(\mathbf{y`}) + |V_1| \\ \mathbf{w}(\mathbf{x`}) &\leq \mathbf{w}(\mathbf{y`}) \end{aligned}$$

Lemma 8.1 then proves that:

$$\mathbf{w}(\mathbf{x`}) \leq \mathbf{w}(\mathbf{y`}) \leq \mathbf{w}(\mathbf{x`})$$

which, as we have already proved **x`** to be optimal, proves the optimality of **y`**.

□

Hence, after reducing the graph in step one we are left with a graph $G' = G[V_{\frac{1}{2}}]$, an optimal all-half primal LPVC(G) solution and a maximum flow in $\overline{G}[V_{\frac{1}{2}}]$. As we found the sets V_0^x, $V_{\frac{1}{2}}^x$ and V_1^x by a single BFS, the time spent on this step is $\mathcal{O}(m+n)$.

8.2.2 Step 2

We now want to find all independent sets with surplus zero to ensure that the all-half solution is the unique primal LP solution before branching.

In the algorithms from sections 6.3 and 7.3 we have iterated through all vertices in the graph, setting their value to one and then re-computed the primal LP in order to find surplus zero sets. If the value of the new primal after fixing a vertex to one was the same as the old, a non all-half optimal primal solution was found. If no vertex gave such a value, the all-half was the unique optimal primal solution. Fixing vertices to one and re-computing the whole primal solution gave rise to the big polynomial part of the running time. In this algorithm all the re-computation of the primal LP is avoided, exploiting properties in the flow network and lemma 8.15. We will see that finding the surplus zero sets in G is the same as finding strongly connected components in the residual graph of $\overline{G}$ with certain properties.

Lemma 8.16. *Let G be a graph in which the all-half primal LP solution is optimal and let $\overline{G}$ be its corresponding network with a maximum flow f^*. For a subset $\overline{S} \subseteq \overline{V}$ in the network, the following statements are equivalent:*

1. *There are no edges going out from $\overline{S}$ into $\overline{V} \setminus \overline{S}$ in $\overline{G_{f^*}}$*
2. *$N(S_L) = S_R$ and $|S_L| = |S_R|$*

Proof. As the all-half vector is an optimal primal solution we know from corollary 8.11 that there exists a flow of amount $|V(G)|$ in $\overline{G}_{f^*}$. This means that there is a unit flow running through all the vertices in $\overline{V}$. We partition

the edges leaving $\overline{S}$ in G_{f^*} into two sets $\overline{E_L}$ and $\overline{E_R}$ according to their start-point, i.e.:

$$\overline{E_L} = \{(l_u, r_v) : l_u \in \overline{L}, r_v \in \overline{R}, (l_u, r_v) \in E(G_{f^*}), l_u \in \overline{S} \text{ and } r_v \notin \overline{S}\}$$
$$\overline{E_R} = \{(r_u, l_v) : r_u \in \overline{R}, l_v \in \overline{L}, (r_u, l_v) \in E(G_{f^*}), r_u \in \overline{S} \text{ and } l_v \notin \overline{S}\}$$

Notice that the only way we can have an edge (r_u, l_v) in the residual graph going from $\overline{R}$ to $\overline{L}$ is if there is flow going through the opposite edge (l_v, r_u). We can now prove the equivalence.

- $1 \Rightarrow 2$

  Assuming 1 is true, we know that $\overline{E_L} = \overline{E_R} = \emptyset$. It is easy to see from $\overline{E_L} = \emptyset$ that all the flow running out of $\overline{S} \cap \overline{L}$ has to flow into $\overline{S} \cap \overline{R}$, and, as $\overline{E_R} = \emptyset$, all flow running in to $\overline{S} \cap \overline{R}$ have to come from $\overline{S} \cap \overline{L}$. The amount of flow running out of $\overline{S} \cap \overline{L} = |S_L|$ and the amount of flow running in to $\overline{S} \cap \overline{R} = |S_R|$, implying $|S_L| = |S_R|$.

  From $\overline{E_L} = \emptyset$ we can also see that $N(S_L) \subseteq S_R$. As there is an integer flow from $\overline{S} \cap \overline{L}$ saturating all vertices in $\overline{S} \cap \overline{R}$, $S_R \subseteq N(S_L)$ and thus $N(S_L) = S_R$.

- $2 \Rightarrow 1$

  Assuming 2 is true, we need to show this implies $\overline{E_L} = \overline{E_R} = \emptyset$. $\overline{E_L} = \emptyset$ follows directly from $N(S_L) = S_R$. As $|S_L| = |S_R|$ all the in-flow of $\overline{S} \cap \overline{R}$ have to come from $\overline{S} \cap \overline{L}$ and thus $\overline{E_R}$ must be empty.

□

As a network is a special directed graph we also have strongly connected components in networks. A strongly connected component is a subset of vertices in the network where all vertices have a directed path to each other, i.e. from any pair of vertices u and v in the component there is a directed path from u to v and from v to u. A tail strongly connected component is a component where none of the vertices have an edge to a vertex outside the component. Notice that any tail strongly connected component in a network will be satisfying the first condition in lemma 8.16.

Lemma 8.17. *If there exists a tail strongly connected component $\overline{S}$ in $\overline{G_{f^*}}$ such that $S_L \cap S_R = \emptyset$, then S_L will be an independent set with surplus zero.*

Proof. As $\overline{S}$ is a tail strongly connected component in $\overline{G}$ we know that part one of the equivalence in 8.16 is satisfied. From this it follows that $\mathrm{N}(S_L) = S_R$ and $|S_L| = |S_R|$. That S_L is independent now follows directly from $S_L \cap S_R = \emptyset$. That $\mathbf{surplus}(S_L) = 0$ follows from $\mathrm{N}(S_L) = S_R$ and that $|S_L| = |S_R|$. □

Lemma 8.18. *If there exists an independent set Z with* ***surplus****(Z) = 0 in G, then* $\overline{T} = \overline{L_z} \cup \overline{R_{N(Z)}}$ *will be a tail strongly connected component in* $\overline{G_{f^*}}$ *with* $T_L \cap T_R = \emptyset$.

Proof. Let Z be a minimal independent set with surplus zero. Define $\overline{T} = \overline{L_z} \cup \overline{R_{N(Z)}}$ to be a subset of $\overline{V}$. As **surplus**(Z) = 0, we know that $|Z| = |N(Z)|$ which implies that $|T_L| = |T_R|$. It is also easy to see that $N(T_L) = T_R$ and hence $\overline{T}$ will satisfy both the conditions in part 2 of lemma 8.16. From this we can conclude that no edges goes from $\overline{T}$ to $\overline{V} \setminus \overline{T}$ and thus no edges points out from $\overline{T}$.

We also need to prove that $\overline{T}$ is a strongly connected component. Assume that it is not strongly connected. Then there will exist a subset $\overline{T'} \subset \overline{T}$ such that there are no edges from $\overline{T'}$ to $\overline{T} \setminus \overline{T'}$. As there are no edges from $\overline{T}$ to $\overline{V} \setminus \overline{T}$ there can not be any edges from $\overline{T'}$ to $\overline{V} \setminus \overline{T'}$. Then $\overline{T'}$ will satisfy the first part of lemma 8.16 and hence T_L will be an independent set in G with surplus zero. T_L will also be a proper subset of Z, contradicting the minimality of Z. Hence $\overline{T}$ must be a tail strongly connected component. □

We have now proven a one to one correspondence between independent sets with surplus zero in G and tail strongly connected components $\overline{S}$ in the residual graph where $S_L \cap S_R = \emptyset$. Thus the task of finding all surplus zero sets in G has been reduced to finding tail strongly connected components in $\overline{G}_{f^*}$. We can do this in linear time, as the following lemma shows:

Lemma 8.19. *For a graph G and a maximum flow f* in* $\overline{G}_{f^*}$*, we can find all surplus zero sets of G in linear time.*

Proof. To compute the strongly connected components of $\overline{G}$ we use the famous algorithm by Tarjan [10], which runs in linear time. All connected components are then connected as an Directed Acyclic Graph(DAG) by creating an edge from one component C_i to another component C_j if there is an edge from a vertex in C_i pointing to a vertex in C_j. Then all tail strongly connected components C can be found and if $C_L \cap C_R = \emptyset$ they can be removed from the network and the graph. When a component is removed all other components pointing to this component must be updated by removing the edge pointing to this component. With some care in the implementation this can be done in linear time. □

After all the surplus zero sets have been found and removed from the network we are left with a flow f^* restricted to the remaining edges in the reduced network $\overline{G'}$. From lemma 8.15 we know that this flow will still be optimal for $\overline{G'}$. Thus in linear time we can find and remove all surplus zero sets in G, guaranteeing that the unique optimal primal solution to the reduced instance is the all half vector and that the flow f^* restricted to the reduced instance is maximum.

8.2.3 Step 3

After step two the algorithm has reduced the given instance as much as possible. We now have a graph G in which $\mathbf{x} \equiv \frac{1}{2}$ is the unique primal LP solution and we also have an optimal dual LP solution given as a maximum flow in a flow network $\overline{G}$. In this section, when referring to a graph G we mean a graph in the state that step 2 has been exhaustively executed. From this we start the branching part of the algorithm.

As mentioned, this algorithm follows a simpler branching strategy than the neighborhood strategy used in the previous algorithms. For every edge $(u, v) \in E(G)$ it is easy to see that either u or v has to be in the vertex cover in order to cover this edge.

Lemma 8.20. *For every edge $(u, v) \in E(G)$, either u or v has to be in any vertex cover.*

Proof. If neither u or v is in the vertex cover then clearly this edge is not covered. □

This give rise to the following branching rule:

Branching Rule 2. *For a graph G, select an arbitrary edge e = (u,v) and branch in two cases. In the first branch, include u in the suggested solution, remove u from G and work recursively the new instance $G' = G - u$ with $k' = k - 1$). The other branch is similar only with v. Then we get the following recursion:*

VC(G) iff VC(G-v, k-1) or VC(G-u, k-1)

Proof. The soundness of this branching rule follows directly from lemma 8.20. □

When a vertex v is removed from G its copies must also be removed from the flow network $\overline{G}$ and the flow needs to be updated. As the all-half vector was a unique optimal LP solution to G before we removed v, we know from corollary 8.12 that there was a unit flow running through each vertex in $\overline{V}$. Hence, when removing the copies l_v and r_v of a vertex $v \in V(G)$, we also remove a flow of amount one running through each of these copies and hence the remaining flow is reduced by two.

From corollary 8.13 the maximum amount of flow running through a flow network can be at most $|V(G)|$. After removing a vertex in a branch we thus see that the maximum obtainable flow in the new instance is reduced by one. As removing the two copies reduced the flow by two, we see that the remaining flow is at most one away from the optimal.

Lemma 8.21. *When branching on a graph G with a maximum flow f^* by removing a vertex v, the optimal flow in the new instance can be found in $\mathcal{O}(m + n)$ time.*

Proof. When removing the copies of v in $\overline{G}$, the flow restricted to the new instance is reduced by two. As the new instance has a maximum flow of at most the amount of f^* -1, the gap between the flow restricted to the new instance and the maximum obtainable flow can be at most one. If this gap is one, it can be found by a single BFS search, searching for augmenting paths in the residual graph of the new instance. If no augmenting paths are found the flow is already optimal. □

Thus we see that branching and updating the flow in step 3 can be done in linear time.

8.3 Time analysis

We have this far seen that all the steps in the algorithm can be done in linear time. Now we need to understand how the different steps changes the parameter $\mu(G, k)$.

Steps one and two reduces the graph according to reduction rule 3 which we previously have seen will not change the parameter. In order to upper bound the number of leaves in the search three created by the algorithm, we need to show that the parameter drops after each branch.

First we state what happens with the optimal primal LP after branching:

Lemma 8.22. *When branching according to branching rule 2, the optimal primal LP of the new instance $G' = G - v$ will be bounded by*

$$vc^*(G') \geq vc^*(G) - \tfrac{1}{2}$$

Proof. Assume $vc^*(G') < vc^*(G) - \frac{1}{2}$. Then any optimal primal solution to G' together with $x_v = 1$ would be a feasible optimal primal solution to G, contradicting the fact that $\mathbf{x} \equiv \frac{1}{2}$ is the unique optimal solution. □

From this we can see that the parameter drops by at least one half after each branch:

Lemma 8.23. *After branching according to branching rule 2, the parameter drops by at least one half.*

Proof. $\mu(G', k') = k' - vc^*(G') \leq k - 1 - (vc^*(G) - \frac{1}{2}) = \mu(G, k) - \frac{1}{2}$ □

From this we see that the parameter drops by one half in each branch, giving a $\mathcal{O}(4^\mu)$ upper bound to the number of leaves in the search tree and thus also the total number of vertices in the tree. We can now state the main result in this chapter:

Theorem 8.24. *We can in time $\mathcal{O}(4^\mu(m + n))$ solve Vertex Cover above LP.*

9 Implementation

The algorithms in this thesis have been implemented in Java for testing purposes. What was of interest to us was to see how the three algorithms performed against each other in practice on different types of graphs. Especially interesting was it to see how an FPT algorithm with improved exponential running time performed against an FPT algorithm where the focus had been to improve the polynomial part of the running time.

This section will be a discussion on what kind of implementation decisions there are when implementing these algorithms and some reasoning why some key decisions were made. We will refer to the algorithm presented in section 6 as algorithm 1, the algorithm in section 7 as algorithm 2 and the last algorithm from section 8 as algorithm 3.

9.1 Algorithm 1 and 2

In order to implement algorithm 1 and 2, many design decisions had to be addressed early. One of the main decisions was to decide whether or not to implement an own graph class. After some research on existing graph libraries in Java, the decision was made on implementing an own. This was mainly due to the increased flexibility having an own implementation provided.

9.1.1 Graph representation in memory

When designing a graph class for the vertex cover problem many considerations must be taken into account. First of all, which kind of graph representation will be the most efficient. In order to answer this question it is helpful to analyze which operations on the graph will be needed during the algorithms and how often they will be needed.

Throughout the algorithm vertices will be removed and reinserted into the graph, making vertex deletion and insertion some of the most often performed operations. Graph traversals, both Breadth-First and Depth-First searches, will also play a vital role in all the algorithms. It is therefore important to efficiently be able to delete and insert vertices to the graph and also to be able to traverse all neighbors of a vertex efficiently.

A matrix representation of a graph will offer constant deletion and insertion, but will have a $\mathcal{O}(n)$ running time for going through all neighbors of a vertex. It will also demand more memory, but this will be of less concern as we work on an NP-Hard problem and most likely will not work on very large instances. The advantages of constant deletion and insertion offered by a matrix representation did not outweigh the linear running time when iterating through neighbors. An adjacency list representation of graphs was therefore chosen.

To implement a graph using an adjacency list representation an own vertex class holding references to all its neighbors was created. All vertices have a unique identifier between zero and n-1 and the graph stores these vertices in a map, mapping an identifier to a vertex. The reason behind using a map instead of a linked list was that one of the reduction rules in algorithm 2 added new "auxillary" vertices to the graph. It was desirable to be able to access vertices in the graph by the unique identifier and thus have a dynamic structure where elements could be deleted and inserted easily. A problematic scenario using a linked list was when vertices are deleted. Ideally a vertex should be stored on the index equal to its identifier, but if a vertex in the list is removed, all the vertices after it will be stored on a new index with lower value. This scenario was especially present as there were no guarantees on how many new "auxillary" vertices could be added during a run of the algorithm and how many of these could be present at the same time. A solution could be to store the vertices in an array where each index in the array corresponded to a unique vertex. The problem then is, how large must this array be in order to take into account the new added auxillary vertices and how to expand the array when needed?

A map between identifiers and vertices was therefore chosen for simplicity when adding and deleting vertices. The choice of a mapping as the data structure comes with an efficiency cost though. Operations like look up, deletion and insertion will in a map take $\mathcal{O}(\log n)$ time. This will not violate the desired polynomial running time in the two first algorithms, but because of the amount of inserts and deletions executed throughout the algorithms, it is suspected that it will add a substantial constant factor to the running time. Having a log factor in look ups, insertions and deletions would violate an algorithm trying to achieve a linear running time, but as this graph structure was only used in the polynomial algorithms, this was not a problem.

9.1.2 Sub structures represented in memory

To simplify the implementation a lot of smaller structures were implemented as own classes. These included augmenting paths, neighborhoods, deleted vertices, vertex covers and reductions. These structures made implementation easier as the different classes could have specific behavior that would be called in different situations. F.ex. the vertex cover class has a method that takes as input a graph and checks if the vertex cover is valid for this graph.

In the classes representing reductions two lists of vertices were stored. One for the vertices in V_0 that could be removed from the graph and one for the vertices in V_1 that could be included in the vertex cover and then removed from the graph. Having these classes simplified storing the state of these vertices while searching for a solution. It was important to store the state of the vertices removed by a reduction rule as later, if no solution was

found in either of the branches, it would be necessary to reinsert them to the graph to get the graph back to its original state.

9.1.3 Hopcroft-Karp

Hopcroft-Karp was needed as a subroutine for finding optimal LPVC(G) solutions in both algorithm 1 and 2. It was implemented as an own class which had a method taking as input a bipartite graph and gave a matching as output. This approach made it easy to reuse the code in both algorithms and it is also available to be used for future implementations of algorithms needing maximum matchings in bipartite graphs.

9.1.4 Considerations

When exhaustively searching for a solution in the branching algorithm, vertex removal and insertions will be performed very often. This happens both when reductions are found and removed from the graph and also when branching on including either a vertex or its neighborhood in the vertex cover. One of the most important things to consider when implementing a recursive branching algorithm is that operations deep in a branch must not affect the state of an instance longer up in the search tree, i.e. when returning from a branch, the instance in the parent node should have exactly the same state as before it started branching. We can say that the branching method has to comply with a contract saying that the instance given as a parameter to the method has to be in the exact same state after the method call returns.

There were two strategies which were considered to make the branching methods comply to this contract. The first was to make copies of the instance at the beginning of the method so each branch would work recursively on a copy of the instance. The good thing about this approach is that it is very easy for the method to comply with the contract. Whatever operations that will be performed in the branch will only be done on the copy of the instance. It is important to be aware that copying in Java often only copies the reference and that the new copy then will point to the same object. Thus, to use this strategy, a copy method would be needed for the instance that would make a deep copy. A deep copy is a new object in the exact same state as the previous with new references to new copies of every object in the original object. This copy method would only use linear time, thus not violating the running time of any of the algorithms, but, as the algorithm would have to copy the entire graph on each node in the search tree, it was feared to add a significant constant factor to the running time. This strategy was therefor not used.

The other strategy considered, and implemented, was to remove only local parts of the graph before branching, and then reinsert them before the

method returns. It is once again important to store the removed parts in a way that makes it easy to insert them so the state of the graph returns to exactly what it was before the part was removed. This was solved by creating special book keeping objects that saved the exact state of the part removed. Thus no operations further down the branch could affect the book keeping objects and it was easy to reinsert it to the graph after both branches had returned.

9.1.5 Reduction rules

All the reduction rules were implemented in own classes for easy reuse of code. Each reduction rule has a method that takes as input a graph, removes the part that can be removed according to the reduction rule and returns a reduction object which stores the parts removed in the state they were before removal. All reduction rules implements a simple interface which allows them to be referenced by this interface but to also have specific behavior depending on the reduction rule they implement.

By this approach new reduction rules can easily be added to a branching algorithm and modifications and improvements can be performed in the specific classes, simplifying maintenance.

9.2 Algorithm 3

In algorithm 3 flow networks and maximum flows was used for finding optimal solutions to LPVC(G). It was therefore necessary with an implementation of a directed graph, which represented a network. A maximum flow algorithm and also an algorithm for finding strongly connected components in a directed graph was needed as sub routines for the main algorithm.

9.2.1 Network

In this algorithm a graph structure is no longer needed during the algorithm as all the operations and updates can be performed on the network created from the graph. The network implemented is not a generalized network, but rather a special class that takes a graph as input in the constructor and creates a network according to definition 8.5.

In this algorithm the only reduction rule applied is reduction rule 3. There is therefore no worries when creating the network that new vertices will later be added, which was a concern when creating the graph class. The network therefore store its vertices in two arrays, one for the left partition and one for the right partition, making look ups, deletions and insertions really easy.

Again, some of the most used operations during the algorithm will be deletions and insertions. This time, instead of deleting vertices from the network, they are marked as being inactive. This approach simplifies the

deletion of a vertex to only switching a flag in it, but also comes with an efficiency cost as every time a graph traversal is done, the neighborhood if a vertex may consist of inactive vertices.

10 Comparison

10.1 Graph classes

In order to test the performance of each algorithm against each other we have implemented three different graph generators which generates graphs with different properties. For the first type no structural properties are known and we have no expectations of how the algorithms run. The second type will consist of many cliques which makes them really dense in clusters, but the clusters will be sparsely connected. They will also have relatively large vertex covers. The last type of graphs generated are planar grid graphs with large matchings. Therefore it is expected that the algorithms will be efficient on these graphs.

10.1.1 Random Graphs

In this generator the desired number of vertices(n) and edges(m) are given as input. A new SimpleGraph instance is then constructed with vertices labeled from zero to n-1. Then m edges will be pseudo randomly added. To find a pseudo random edge, the generator uses the method nextInt(int n) from the java class Random to pseudo randomly pick a start-point and an end-point of the edge in the range from zero to n-1. This edge will then be added to the graph. If the edge is already present or the edge is a self loop, the implementation of SimpleGraph will ignore the request of adding this edge and a new psuedo random edge will be found. This process is repeated until m edges have been successfully added to the graph.

We know no structural properties of the graphs generated by this generator other than the density of the graph. The graphs generated may be connected or consist of several connected components. It can also have isolated vertices. It is therefore interesting to test the algorithms on graphs of this type.

10.1.2 Clique Graphs

We call the second graph class we generate, somewhat inaccurately, for Clique Graphs. The reason behind this name is that the generator creates a number of cliques and then connects them by a pseudo random number of edges. It does this by taking as input the desired numbers of vertices(n) in the graph and then a parameter k, where k decides the maximum size allowed for a clique in the graph. It then proceeds to find l cliques of size

between one and k such that the sum of all the cliques are n. The cliques get an index between zero and l and all consecutive cliques are connected by a pseudo randomly picked number of edges. Thus the graph is like a path of cliques that can be connected with multiple edges.

The structural properties of these graphs makes it easy to determine a lower bound on the number of vertices in a vertex cover for them. Let G be a Clique Graph generated by this generator with cliques $Q = \{Q_1, ..., Q_l\}$. The size of each $Q_i \in Q$ is less than k and the sum of the sizes of all the cliques are n. It is easy to see that a vertex cover for a clique of size k must have k-1 vertices. This helps us create the lower bound $\sum_{Q_i \in Q} |Q_i| - 1$ on the number of vertices needed for a vertex cover for G.

This lower bound was a motivation for creating these types of graphs as it made early testing of vertex cover algorithms easier. If an algorithm returned a vertex cover of size less than the lower bound it obviously was wrong. These graphs also have a relatively large vertex cover and it is therefore expected that it will be time consuming to find vertex covers for them, even for small graphs. Also, as the algorithms we have implemented tries to exploit a parameter that will make vertex cover instances with high solution size more tractable, these graphs are interesting.

10.1.3 Grid Graphs

The third graph class we generate we call Grid Graphs. This generator takes as input the desired width(w) and height(h) of a grid and creates a new SimpleGraph instance with $w \cdot h$ vertices. It then connects the vertices as a grid before it removes a pseudo random number of edges in the graph.

These graphs will be planar and quite sparse. The also contain large matchings, so it is expected that the algorithms will be efficient on these graph types.

10.2 Testing

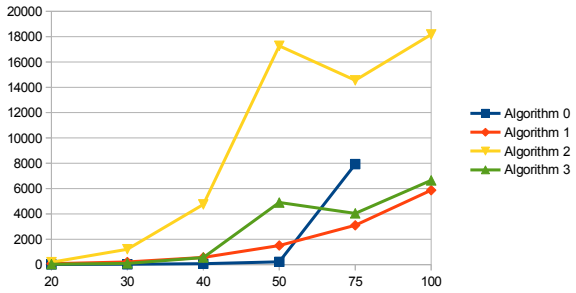
As we began with this thesis a case study of Vertex Cover was published by Takuya Akiba and Yoichi Iwata [1]. It would be desirable to test our implementation against their results, but, as this implementation is done in Java and the tests are performed on a personal laptop, such a comparison would not make much sense. Instead we focused on comparing the algorithms to a naive neighborhood branching algorithm which did not use any reduction rules to see if the algorithms presented in this thesis would perform better. We call this naive algorithm for algorithm 0.

The way the algorithms are tested is by generating many graphs of each size in the different graph generators and then finding a minimum vertex cover fpr them. The efficiency will be measured in the average time it takes for an algorithm to find a minimum vertex cover of a graph of this size. The

algorithms searches for a minimum vertex cover by starting with k equal to one and then increasing it by one until it finds a valid vertex cover.

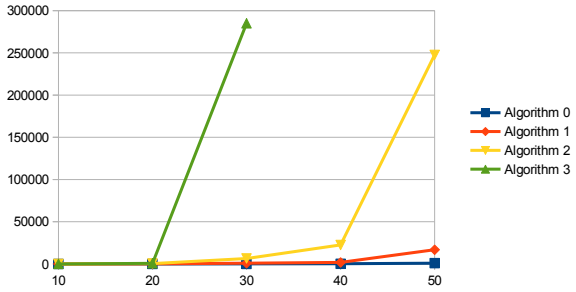
10.2.1 Random Graphs

To test this class, graphs of size 20 to 100 have been generated and then tested. The x-axis shows the number of vertices in the graph and the y-axis gives the time in milliseconds. Algorithm 0 was not run on instances of size 100 because it took too long to terminate:



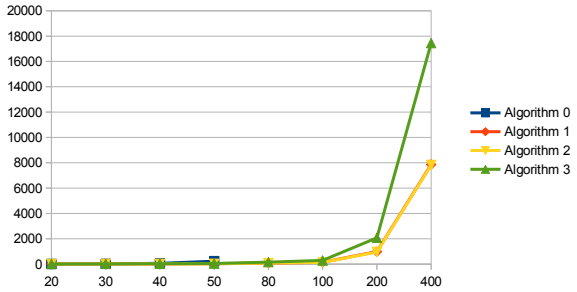
10.2.2 Clique Graphs

The algorithms where substantially slower for these kinds of graphs and only graphs of size 10 to 50 were tested. Again the x-axis shows the size of the instances while the y-axis shows the time in milliseconds. In this case it was Algorithm 0 which could handle the largest cases most efficiently:



10.2.3 Grid Graphs

In this class larger instances with up to 400 vertices was generated. Algorithm 0 was not run on instances larger than 50 vertices as it took too long for it to terminate. As before, the x-axis shows the size of the instances while the y-axis shows the time in milliseconds:



10.2.4 Discussion

The results presented in the previous sections are of little value besides being pointers on how the different algorithms behave on different types of graphs. This is due to the small sample space they are from and that there was no time to run the algorithms on larger instances.

We can nonetheless see some tendencies from the results. Especially the results for Grid Graphs were as expected. These graphs are sparse and each vertex has a degree of four or less. The neighborhood branching strategy will therefore never lead to big parameter drops in its branches, something which could speed up the practical running time considerably. These graphs also have a large maximum matching and a vertex cover close to the matching in size, so the above guarantee parameter can be expected to be small and thus the above guarantee parameter will be a reasonable one.

For Clique Graphs the effect of the neighborhood branching strategy will be opposite. In these graphs one can expect that a vertex has a big neighborhood, thus leading to big drops in the parameter. These graphs also have relatively large vertex covers, so the gap between an optimal LPVC(G) and k will still be large. This can explain why the naive algorithm performed the best on these graphs.

For a more thorough examination of how well the algorithms perform it could be wise to also monitor other behavior than just the time it takes the algorithms to terminates. The number of branches throughout an algorithm and the number of reductions and the size of the reductions can tell us more about the actual value of the reduction rules.

11 Comment on working with the thesis

The first part of working with this thesis was to get familiarized with the Vertex Cover above LP problem and all the theory related to Linear Programming and the above guarantee parameter. This was done by reading and understanding the first two algorithms.

Then time was spent to fully understand the linear time algorithm from [6]. In this paper the authors focused on a slightly different version of the problem, namely Vertex Cover above LP in weighted graphs. An important goal of this thesis was to present the three algorithms in a concise way and an important part of this was to rewrite and simplify the results from [6] to the unweighted case. This constituted a substantial amount of work on this thesis.

The three algorithms have been published in three different papers with different authors and inevitably they will use different terminology and language. We wanted to presents all the algorithms in a unified language using the same terminology throughout. We have also tried to highlight the similarities between the different algorithms and believe that this thesis will therefore be a good starting point for anyone interested in this problem.

The second part of this thesis was to actually implement the algorithms and test them on different graphs. Due to inexperience with programming this was a time consuming and difficult part of the thesis. In the end all algorithms run correctly, to the best of our knowledge, but are far from being optimized. The experimental results presented are therefore of little empirical value.

References

[1] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *CoRR*, abs/1411.2680, 2014.

[2] Claude Berge. "two theorems in graph theory.". *Proceedings of the National Academy of Sciences of the United States of America 43.9*, 1957.

[3] Jianer Chen, Iyad A Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40):3736–3756, 2010.

[4] M. Cygan, F. V. Fomin, D. Lokshtanov, Ł. Kowalik, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015. In press.

[5] P. Hall†. Classic papers in combinatorics. pages 58–62, 1987.

[6] Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. Linear-time FPT algorithms via network flow. *CoRR*, abs/1307.4927, 2013.

[7] Richard M. Karp John E. Hopcroft. "an n5/2 algorithm for maximum matchings in bipartite graphs". *SIAM Journal of Computing*, 1973.

[8] Daniel Lokshtanov, N. S. Narayanaswamy, Venkatesh Raman, M. S. Ramanujan, and Saket Saurabh. Faster parameterized algorithms using linear programming. *CoRR*, abs/1203.0833, 2012.

[9] G.L. Nemhauser and Jr. Trotter, L.E. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

[10] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[11] Robert J. Vanderbei. Linear programming: Foundations and extensions, 1996.